
PyTorch


PyTorch

What is PyTorch?

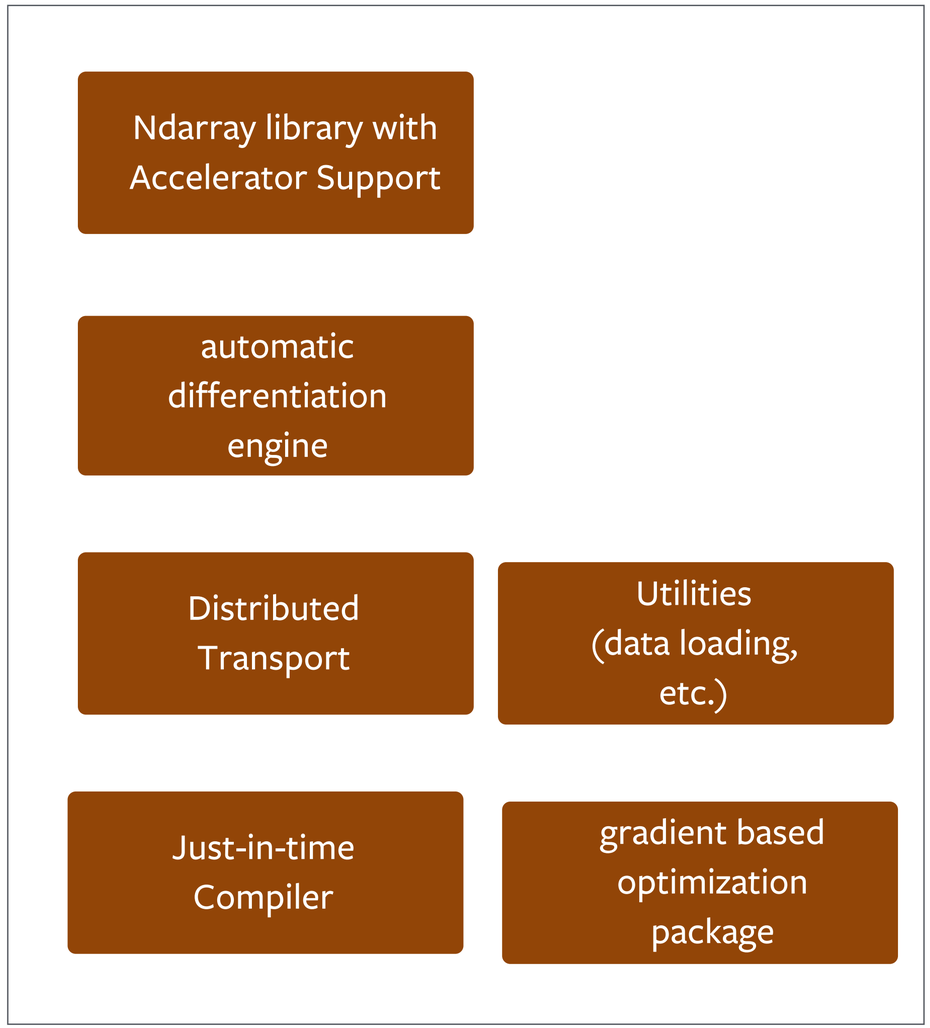
Ndarray library with Accelerator Support	automatic differentiation engine	gradient based optimization package	Utilities (data loading, etc.)	Distributed Transport	Just-in-time Compiler

Numpy-alternative

Deep Learning

Reinforcement Learning

PyTorch = libtorch + Python bindings



libtorch (C++)

Python Bindings

Utilities (data loading, etc.)

gradient based optimization package

PyTorch (Python)

ndarray / Tensor library

- np.ndarray <-> torch.Tensor
- ~450 operations, similar to numpy
- very fast acceleration on GPUs / TPUs / xPUs

```numpy
# -*- coding: utf-8 -*-
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
```

Numpy

```pytorch
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
```

PyTorch

ndarray / Tensor library

Tensors are similar to NumPy's ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing.

```
from __future__ import print_function
import torch
```

Construct a 5x3 matrix, uninitialized:

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

Out:

```
tensor([[2.0705e-19, 2.0535e-19, 5.0722e-14],
        [7.5019e+28, 4.6114e+24, 6.9983e+28],
        [2.6336e+20, 4.9664e+28, 4.6114e+24],
        [2.7675e+20, 5.2839e-11, 6.8571e+22],
```

ndarray / Tensor library

Construct a randomly initialized matrix

```
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

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

Out:

```
torch.Size([5, 3])
```

ndarray / Tensor library

You can use standard numpy-like indexing with all bells and whistles!

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

Out:

```
0.7231
 0.1244
 0.2700
 0.9455
 0.8230
[torch.FloatTensor of size 5]
```

ndarray / Tensor library

```
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

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

Out:

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

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

Out:

```
[ 1.  1.  1.  1.  1.]
```

NumPy bridge

Converting torch Tensor to numpy Array

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

Out:

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

Zero memory-copy
very efficient

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

Out:

```
[ 1.  1.  1.  1.  1.]
```

Seamless GPU Tensors

```
# let us run this cell only if CUDA is available
# We will use ``torch.device`` objects to move tensors in and out of GPU
if torch.cuda.is_available():
    device = torch.device("cuda")          # a CUDA device object
    y = torch.ones_like(x, device=device)  # directly create a tensor on GPU
    x = x.to(device)                       # or just use strings ``.to("cuda")``
    z = x + y
    print(z)
    print(z.to("cpu", torch.double))       # ``.to`` can also change dtype together!
```

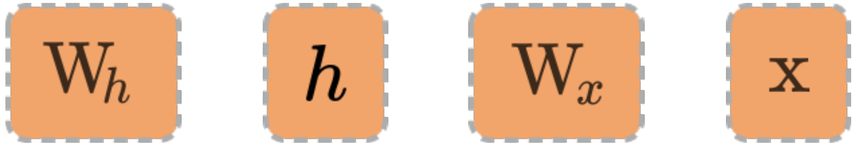
Out:

```
tensor([0.3882], device='cuda:0')
tensor([0.3882], dtype=torch.float64)
```

automatic differentiation engine

for deep learning and reinforcement learning

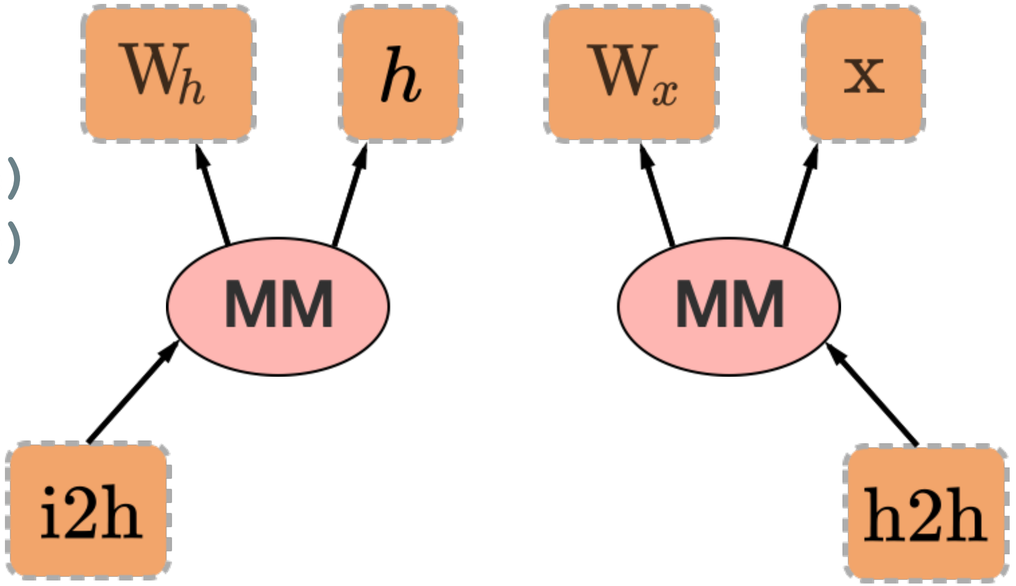
PyTorch Autograd




```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```

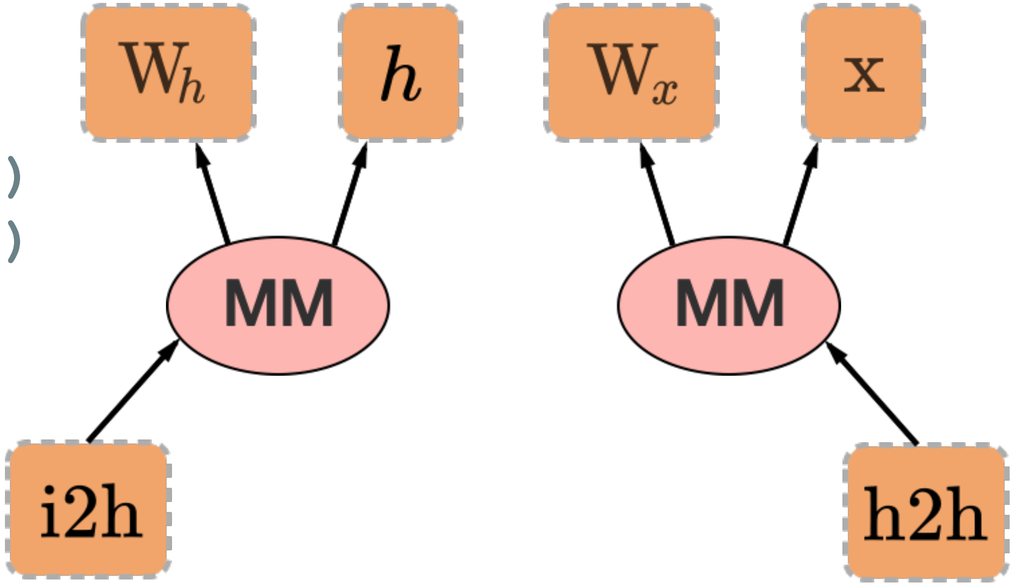
PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)


i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
```



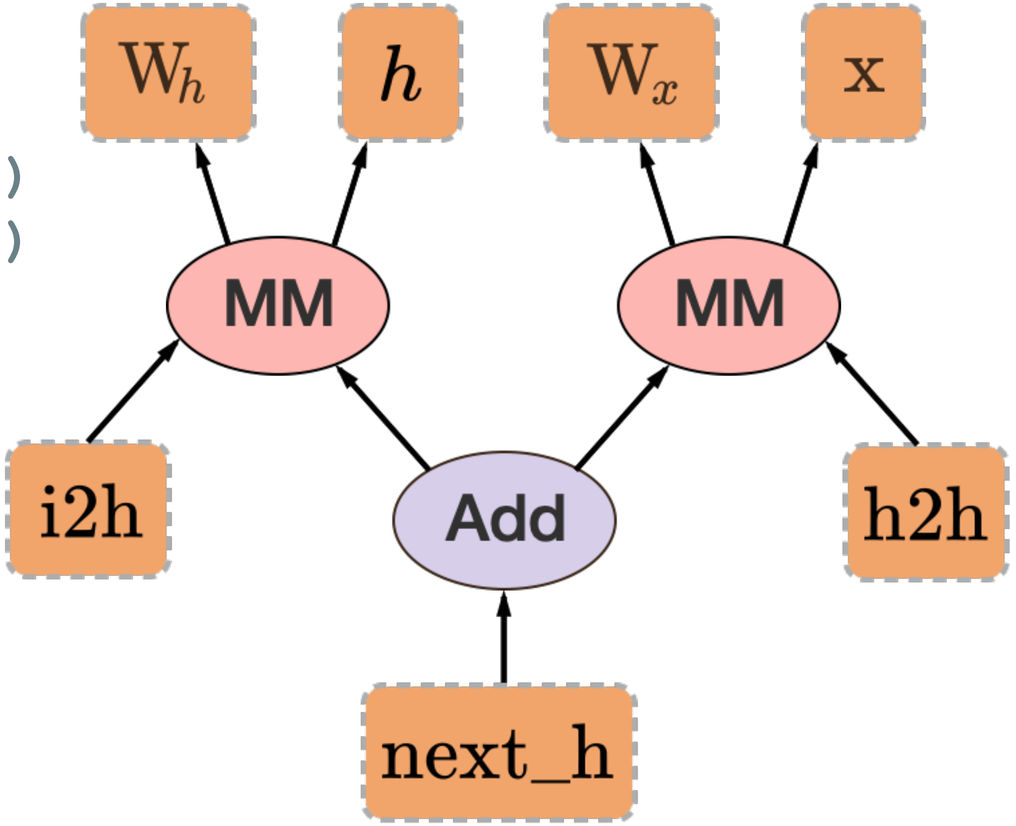
PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)


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



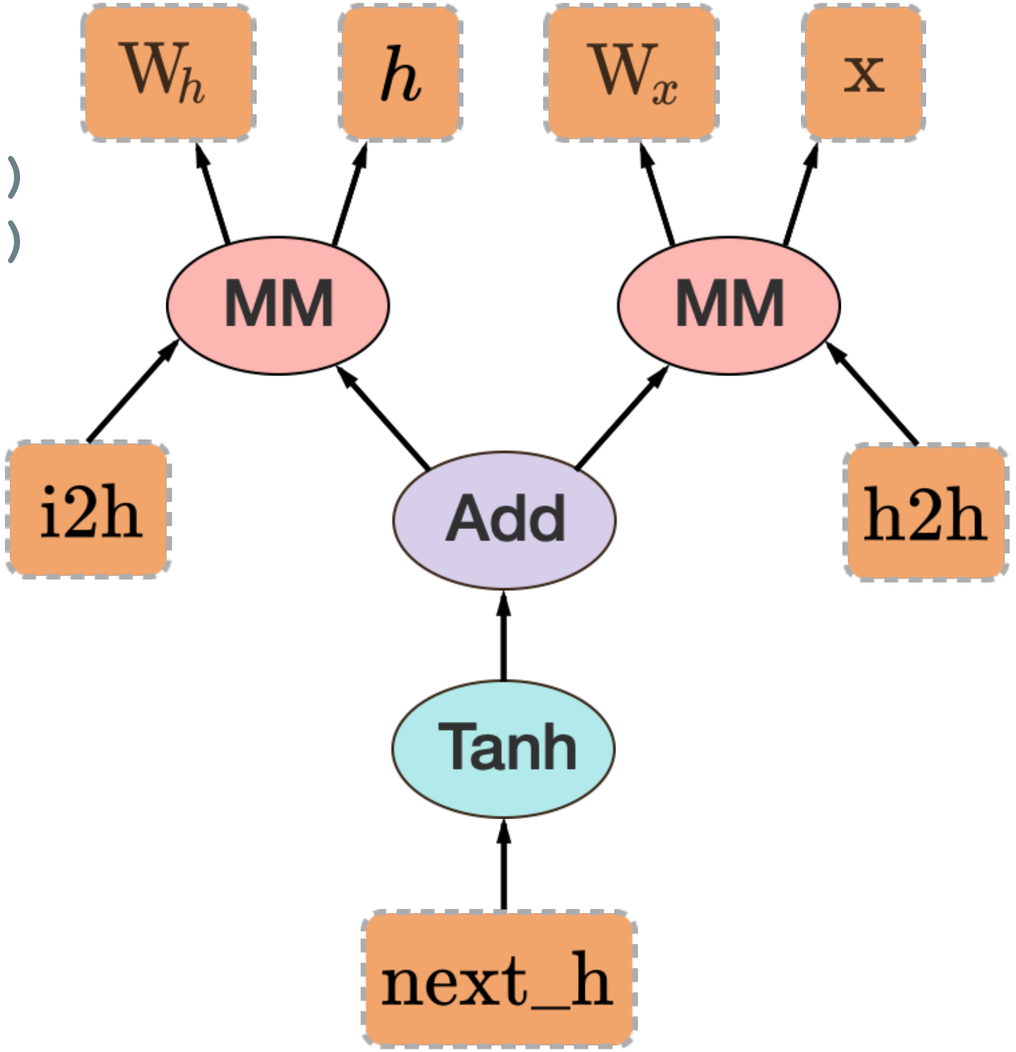
PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)


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



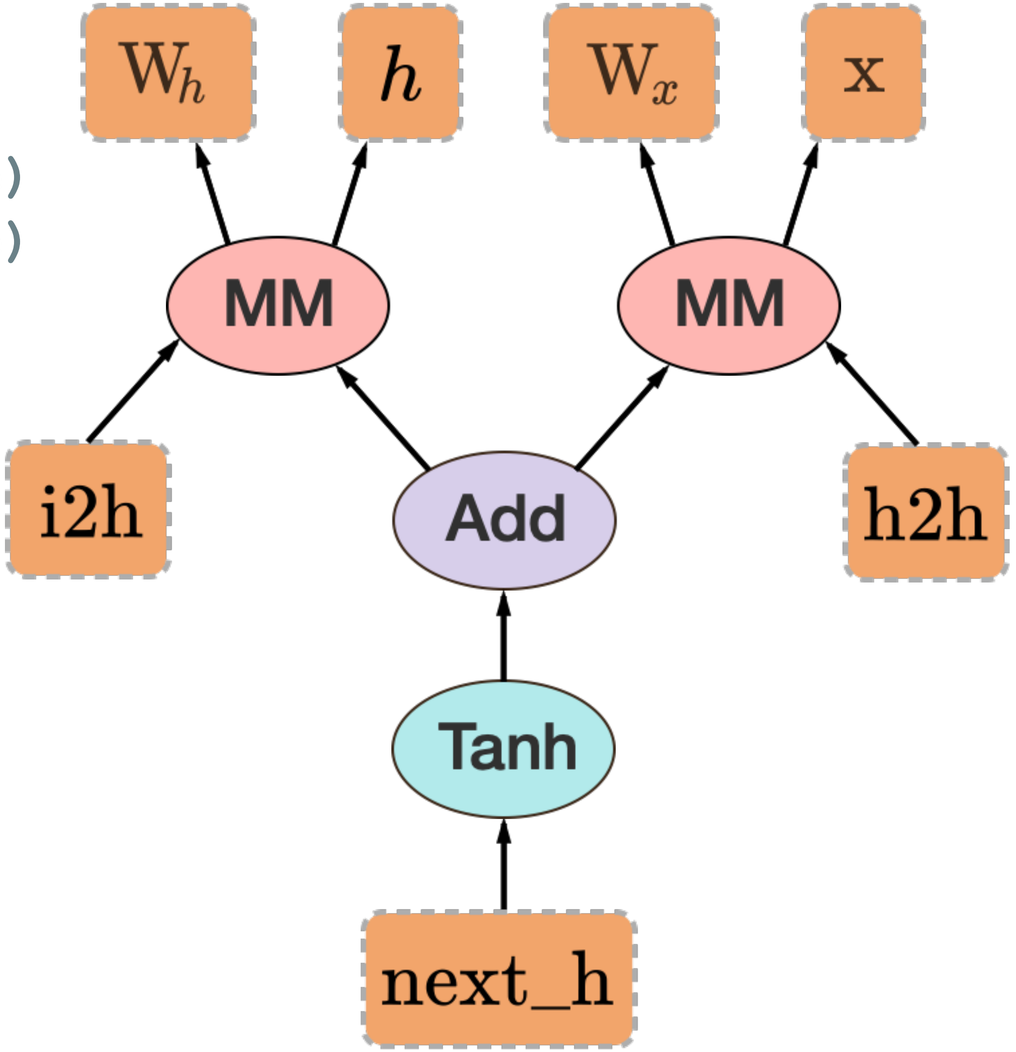
PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)


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



PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)


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

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)


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))
```

- Higher order Gradients
- Mutability

neural networks

Neural Networks

```
1   class Net(nn.Module):
2       def __init__(self):
3           super(Net, self).__init__()
4           self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
5           self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
6           self.conv2_drop = nn.Dropout2d()
7           self.fc1 = nn.Linear(320, 50)
8           self.fc2 = nn.Linear(50, 10)
9
10      def forward(self, x):
11          x = F.relu(F.max_pool2d(self.conv1(x), 2))
12          x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
13          x = x.view(-1, 320)
14          x = F.relu(self.fc1(x))
15          x = F.dropout(x, training=self.training)
16          x = self.fc2(x)
17          return F.log_softmax(x)
18
19  model = Net()
20  input = Variable(torch.randn(10, 20))
21  output = model(input)
```

Neural Networks

```
1   class Net(nn.Module):
2       def __init__(self):
3           super(Net, self).__init__()
4           self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
5           self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
6           self.conv2_drop = nn.Dropout2d()
7           self.fc1 = nn.Linear(320, 50)
8           self.fc2 = nn.Linear(50, 10)
9
10      def forward(self, x):
11          x = F.relu(F.max_pool2d(self.conv1(x), 2))
12          x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
13          x = x.view(-1, 320)
14          x = F.relu(self.fc1(x))
15          x = F.dropout(x, training=self.training)
16          x = self.fc2(x)
17          return F.log_softmax(x)
18
19  model = Net()
20  input = Variable(torch.randn(10, 20))
21  output = model(input)
```

Neural Networks

```
1   class Net(nn.Module):
2       def __init__(self):
3           super(Net, self).__init__()
4           self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
5           self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
6           self.conv2_drop = nn.Dropout2d()
7           self.fc1 = nn.Linear(320, 50)
8           self.fc2 = nn.Linear(50, 10)
9
10      def forward(self, x):
11          x = F.relu(F.max_pool2d(self.conv1(x), 2))
12          x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
13          x = x.view(-1, 320)
14          x = F.relu(self.fc1(x))
15          x = F.dropout(x, training=self.training)
16          x = self.fc2(x)
17          return F.log_softmax(x)
18
19  model = Net()
20  input = Variable(torch.randn(10, 20))
21  output = model(input)
```

Neural Networks

```
1   class Net(nn.Module):
2       def __init__(self):
3           super(Net, self).__init__()
4           self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
5           self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
6           self.conv2_drop = nn.Dropout2d()
7           self.fc1 = nn.Linear(320, 50)
8           self.fc2 = nn.Linear(50, 10)
9
10      def forward(self, x):
11          x = F.relu(F.max_pool2d(self.conv1(x), 2))
12          x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
13          x = x.view(-1, 320)
14          x = F.relu(self.fc1(x))
15          x = F.dropout(x, training=self.training)
16          x = self.fc2(x)
17          return F.log_softmax(x)
18
19  model = Net()
20  input = Variable(torch.randn(10, 20))
21  output = model(input)
```

In Eager Mode: Python is the Interpreter / VM

Neural Networks

In Graph Mode: torch.jit is the Interpreter / VM

```
1   class Net(nn.Module):
2       def __init__(self):
3           super(Net, self).__init__()
4           self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
5           self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
6           self.conv2_drop = nn.Dropout2d()
7           self.fc1 = nn.Linear(320, 50)
8           self.fc2 = nn.Linear(50, 10)
9
10      def forward(self, x):
11          x = F.relu(F.max_pool2d(self.conv1(x), 2))
12          x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
13          x = x.view(-1, 320)
14          x = F.relu(self.fc1(x))
15          x = F.dropout(x, training=self.training)
16          x = self.fc2(x)
17          return F.log_softmax(x)
18
19  model = Net()
20  input = Variable(torch.randn(10, 20))
21  output = model(input)
```

Optimization package

SGD, Adagrad, RMSProp, LBFGS, etc.

```
model = Net()
optimizer = optim.SGD(model.parameters())

model.train()
for data, target in enumerate(training_data):
    optimizer.zero_grad()
    # run the model forward
    output = model(data)
    loss = F.nll_loss(output, target)

    # gradients propagated to model.parameters()
    loss.backward()

    # step uses computed gradients to update
parameters
    optimizer.step()
```

Distributed

C10D LIBRARY

DESIGN AND FEATURES

— Backends: Gloo, NCCL, MPI
— Fully async collectives for backends
— Both Python and C++ APIs
— CUDA streams for parallelism
— Future: Fault tolerance with elasticity
— Future: Integration with APEX

torch.distributed

SYNC MODE

```
# Backward compatible synchronous collective op
torch.distributed.all_reduce(tensor, op, group, async_op=False)
```

ASYNC MODE

```
# New asynchronous collective op
work = torch.distributed.all_reduce(tensor, op, group, async_op=True)
work.wait()
```

TURN KEY SOLUTION

torch.nn.DistributedDataParallel

JUST WRAP YOUR MODEL

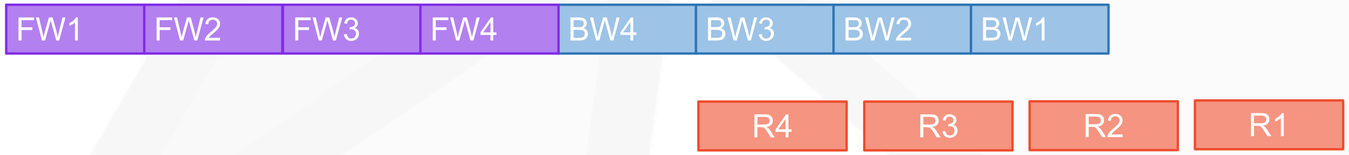
```
torch.distributed.init_process_group(world_size=4, init_method='...')
model = torch.nn.DistributedDataParallel(model)

for epoch in range(max_epochs):
    for data, target in enumerate(training_data):
      output = model(data)
      loss = F.nll_loss(output, target)
      loss.backward()
      optimizer.step()
```

DISTRIBUTED DATA PARALLEL

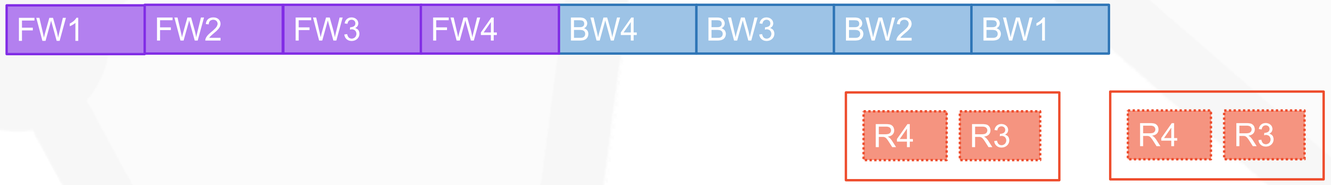
NO OVERLAPPING

An iteration: Forward (FW) -> Backward(BW) -> AllReduce(R)

OVERLAPPING BACKWARD WITH REDUCE

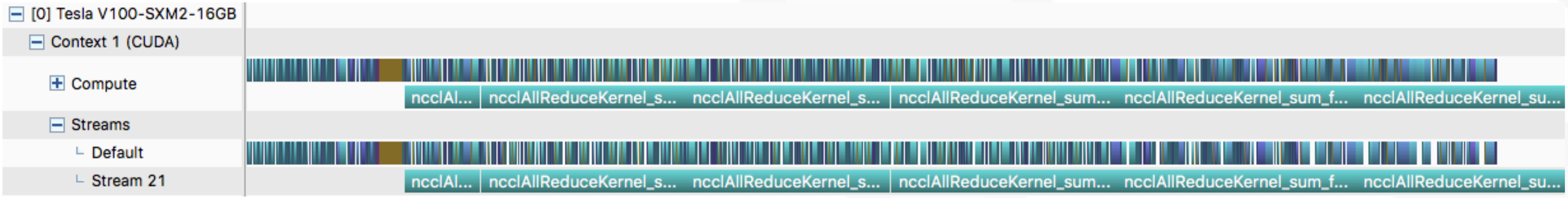


TENSOR COALESCING/ BUCKETING

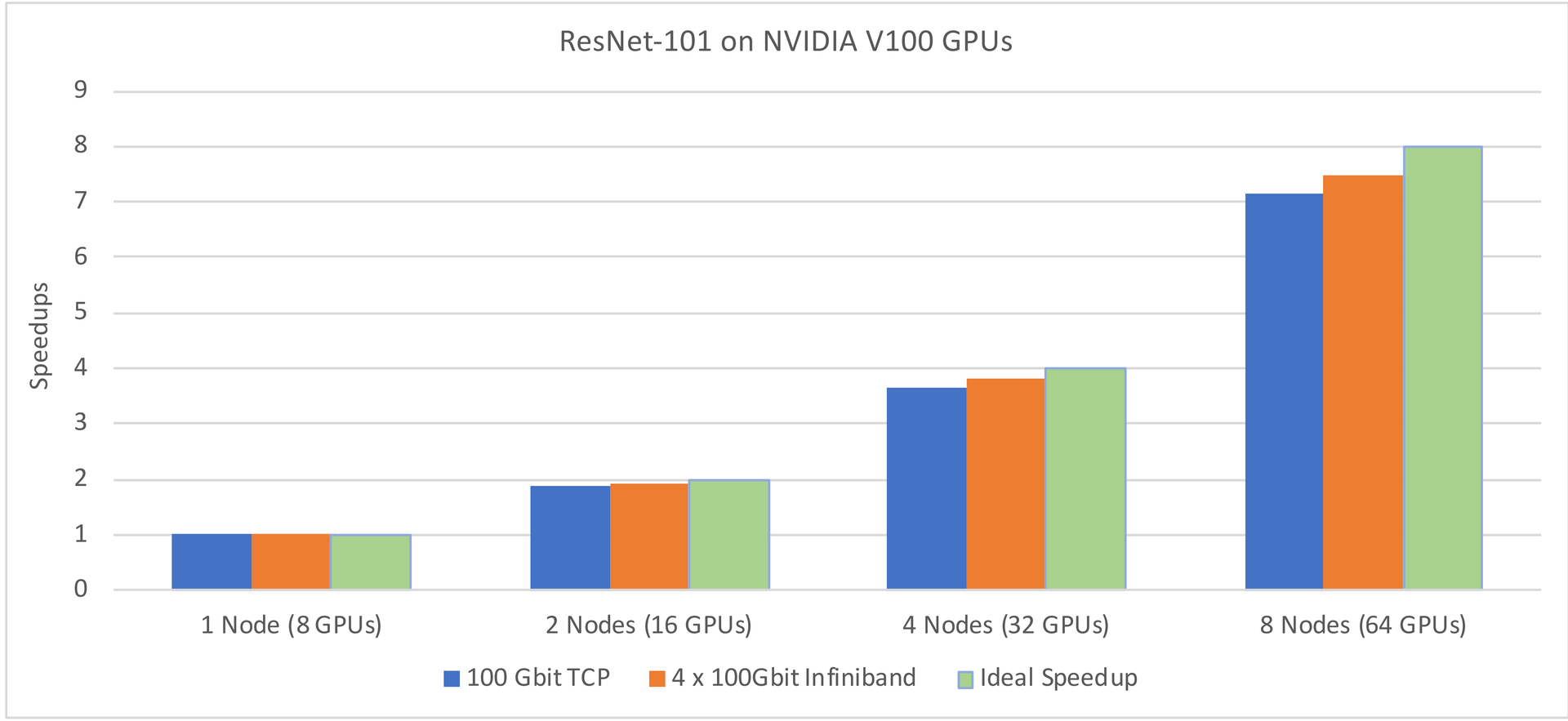


Performance-driven design

- Overlapping BWs with all-reductions
- Coalescing small tensors into buckets
  - A bucket is a big coalesced tensor

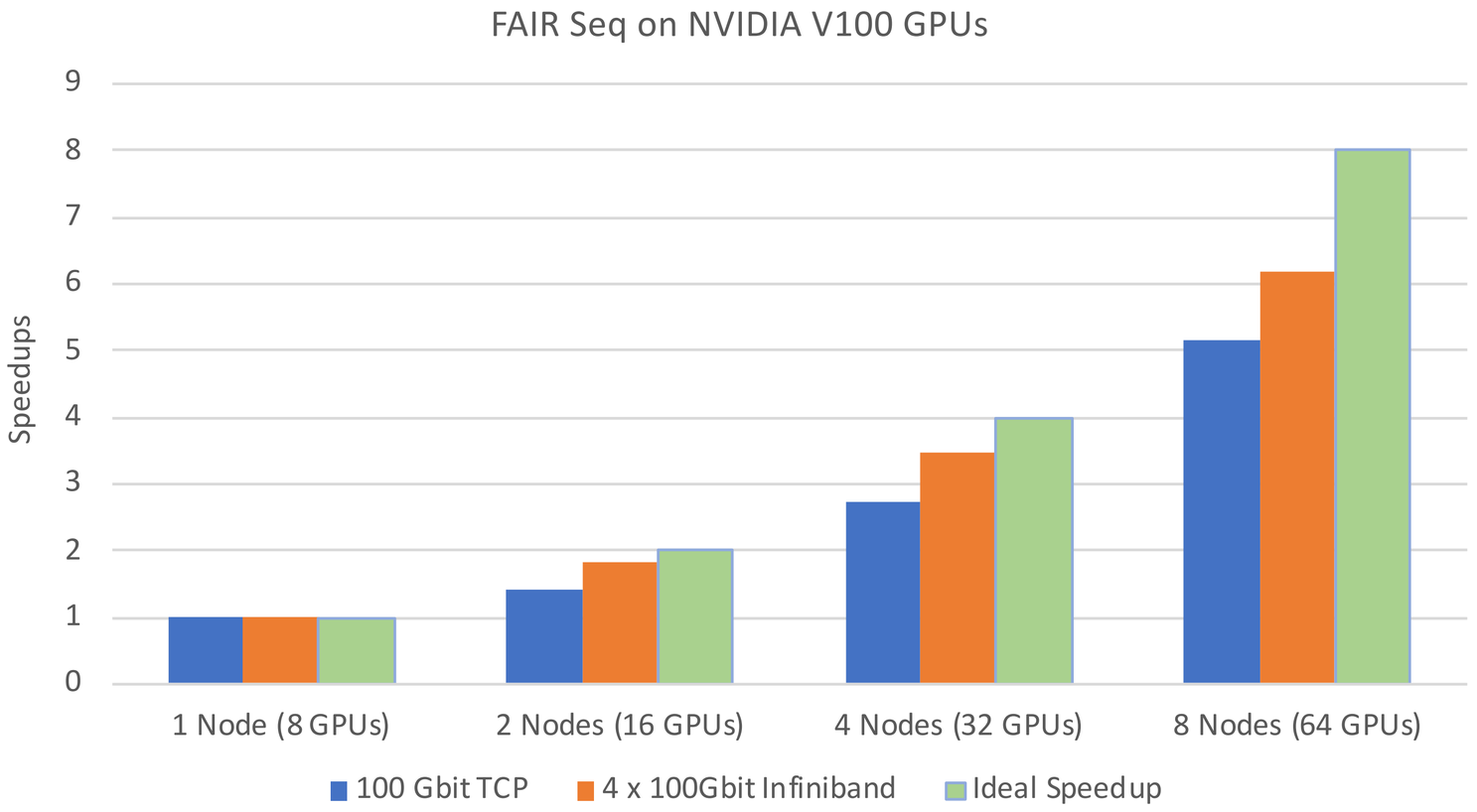


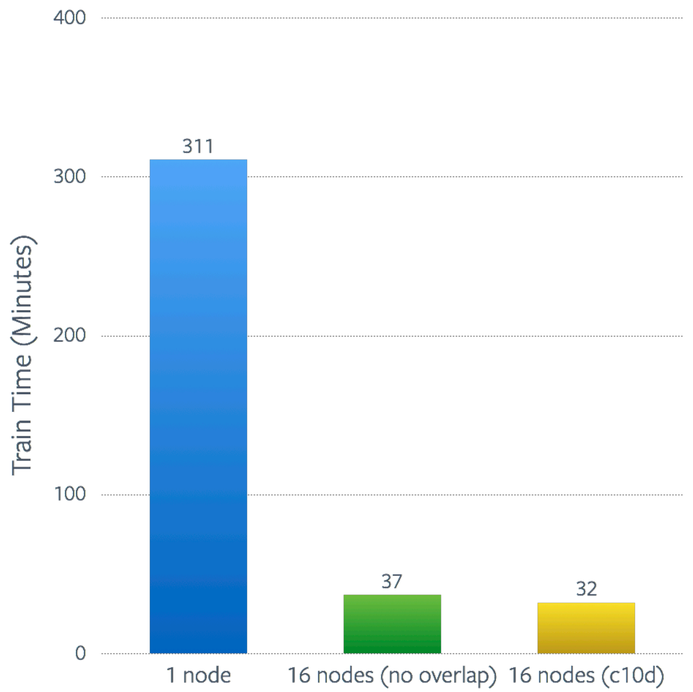
Distributed Training Performance – ResNet101



Distributed Training Performance – FAIR Seq

Bonjour à tous ! → Hello everybody!





- 311 minutes – 32 minutes, by going from 1 to 16 NVIDIA DGX-1 nodes (8 to 128 NVIDIA V100 GPUs)
- 19% performance gain (1.53M – 1.82M Words Per Second on 16 nodes), thanks to c10d DDP overlapping

Data Pipeline

Data Loading / infeed

```
model = Net()
optimizer = optim.SGD(model.parameters())

model.train()
for data, target in enumerate(training_data):
    optimizer.zero_grad()
    # run the model forward
    output = model(data)
    loss = F.nll_loss(output, target)

    # gradients propagated to model.parameters()
    loss.backward()

    # step uses computed gradients to update
parameters
    optimizer.step()
```

Data Loading / infeed

- Provides a Process-pool mechanism
- Give simple classes with a structured API to replicate and parallelize

Data L

- Provides
- Give sim
  paralleliz

```
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
dataloader = torch.utils.data.DataLoader(dataset, batch_size=opt.batchSize,
                                         shuffle=True, num_workers=int(opt.workers))
```

```
                                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
dataloader = torch.utils.data.DataLoader(dataset, batch_size=opt.batchSize,
                                         shuffle=True, num_workers=int(opt.workers))

```

Data Loading / infeed

- Pre-processing is a function of operators:
  - Parallelize them as part of data-loading

  -OR
  - Make them as part of your model

Data Loading / infeed

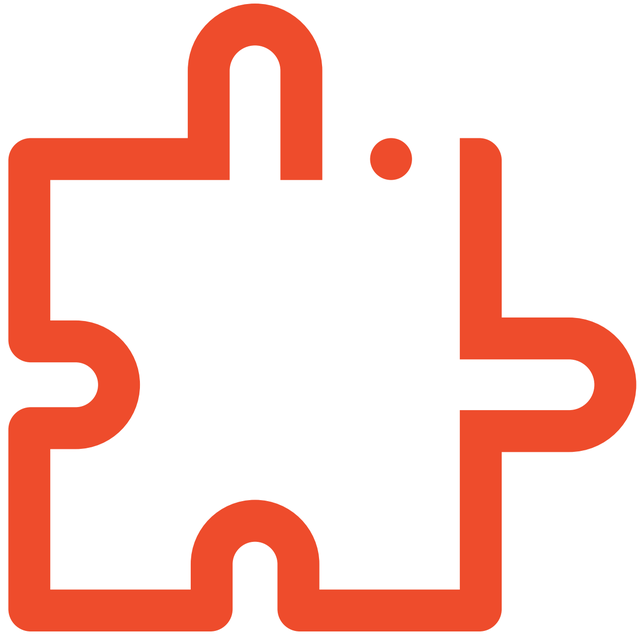
- Pre-processing is a function of operators:

```
def forward(self, x):
    if self.transform_input:
        x_ch0 = torch.unsqueeze(x[:, 0], 1) * (0.229 / 0.5) + (0.485 - 0.5) / 0.5
        x_ch1 = torch.unsqueeze(x[:, 1], 1) * (0.224 / 0.5) + (0.456 - 0.5) / 0.5
        x_ch2 = torch.unsqueeze(x[:, 2], 1) * (0.225 / 0.5) + (0.406 - 0.5) / 0.5
        x = torch.cat((x_ch0, x_ch1, x_ch2), 1)

    # N x 3 x 224 x 224
    x = self.conv1(x)
    # N x 64 x 112 x 112
    x = self.maxpool1(x)
```

C++ as a first-class citizen

C++ EXTENSIONS

The power of C++, CUDA and ATen
in imperative PyTorch models



```cpp
#include <torch/extension.h>
#include <opencv2/opencv.hpp>

at::Tensor compute(at::Tensor x, at::Tensor w) {
  cv::Mat input(x.size(0), x.size(1), CV_32FC1, x.data<float>());
  cv::Mat warp(3, 3, CV_32FC1, w.data<float>());

  cv::Mat output;
  cv::warpPerspective(input, output, warp, {64, 64});

  return torch::from_blob(output.ptr<float>(), {64, 64}).clone();
}

PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
  m.def("compute", &compute);
}
```

C++ EXTENSION

```python
from setuptools import setup
from torch.utils.cpp_extension \
    import BuildExtension, CppExtension

setup(
    name='extension',
    packages=['extension'],
    ext_modules=[CppExtension(
        name='extension',
        sources='extension.cpp',
    )],
    cmdclass=dict(build_ext=BuildExtension))
```

SETUPTOOLS

```python
import torch.utils.cpp_extension

module = torch.cpp_extension.load(
    name='extension',
    sources='extension.cpp',
)

module.compute(...)
```

JIT EXTENSION

```
import torch
import extension

image = torch.randn(128, 128)
warp = torch.randn(3, 3)
output = extension.compute(image, warp)
```

PYTHON INTEGRATION

C++ FRONTEND

The aesthetics of imperative PyTorch for high performance, pure C++ research environments

MISSION

The aesthetics of PyTorch in pure C++

MOTIVATION

Enable research in environments that are . . .

MISSION

The aesthetics of PyTorch in pure C++

VALUES

Enable research in environments that are . . .

LOW LATENCY

BARE METAL

MULTITHREADED

ALREADY C++

torch::nn

NEURAL NETWORKS

torch::optim

OPTIMIZERS

torch::data

DATASETS &
DATA LOADERS

torch::serialize

SERIALIZATION

torch::python

PYTHON INTER-OP

torch::jit

TORCH SCRIPT
INTER-OP

```cpp
#include <torch/torch.h>

struct Net : torch::nn::Module {
  Net() : fc1(8, 64), fc2(64, 1) {
    register_module("fc1", fc1);
    register_module("fc2", fc2);
  }

  torch::Tensor forward(torch::Tensor x) {
    x = torch::relu(fc1->forward(x));
    x = torch::dropout(x, /*p=*/0.5);
    x = torch::sigmoid(fc2->forward(x));
    return x;
  }

  torch::nn::Linear fc1, fc2;
};
```

C++

```python
import torch

class Net(torch.nn.Module):
    def __init__(self):
        self.fc1 = torch.nn.Linear(8, 64)
        self.fc2 = torch.nn.Linear(64, 1)

    def forward(self, x):
        x = torch.relu(self.fc1.forward(x))
        x = torch.dropout(x, p=0.5)
        x = torch.sigmoid(self.fc2.forward(x))
        return x
```

PYTHON

```cpp
Net net;

auto data_loader = torch::data::data_loader(
  torch::data::datasets::MNIST("./data"));

torch::optim::SGD optimizer(net->parameters());

for (size_t epoch = 1; epoch <= 10; ++epoch) {
  for (auto batch : data_loader) {
    optimizer.zero_grad();
    auto prediction = net->forward(batch.data);
    auto loss = torch::nll_loss(prediction,
                                batch.label);
    loss.backward();
    optimizer.step();
  }
  if (epoch % 2 == 0)
    torch::save(net, "net.pt");
}
```

C++

```python
net = Net()

data_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('./data'))

optimizer = torch.optim.SGD(net.parameters())

for epoch in range(1, 11):
    for data, target in data_loader:
        optimizer.zero_grad()
        prediction = net.forward(data)
        loss = F.nll_loss(prediction, target)
        loss.backward()
        optimizer.step()
    if epoch % 2 == 0:
        torch.save(net, "net.pt")
```

PYTHON

```cpp
// Creating the process group with store method
auto store = std::make_shared<FileStore>("test");
ProcessGroupNCCL pg(store, RANK, WORLD_SIZE);

// Kicking off work
for (auto i = 0; i < tensors.size(); ++i) {
  std::vector<at::Tensor> tmp = {tensors[i]};
  works.push_back(pg.allreduce(tmp));
}

for (auto& work : works) {
  work->wait();
}
```

C++

```python
# Creating the process group with store method
store = dist.FileStore("/tmp/test")
pg = dist.ProcessGroupNCCL(store, RANK, WORLD_SIZE)

# Kicking off work
# Assuming that tensors are a list of Tensors
works = []
for tensor in tensors:
    work = pg.allreduce([tensor], opts)
    works.append(work)

for work in works:
    work.wait()
```

PYTHON

torch.jit and production

Production Requirements

Portability

Models should be exportable to a wide variety of environments, from C++ servers to mobile.

Performance

We want to optimize common patterns in neural networks to improve inference latency and throughput.

Production Requirements

How does PyTorch do?

Portability

Models should be exportable to a wide variety of environments, from C++ servers to mobile.

Tight coupling of models to the Python runtime makes exporting difficult.

Performance

We want to optimize common patterns in neural networks to improve inference latency and throughput.

Dynamic graphs mean we can't "know" what will happen.

Why not use a static framework?

A single tool for research to production.

Can put new research results into models quickly.

People want Pythonic.

Flexibility and ease of use is a hard requirement.

PyTorch JIT

We need a system that can:

1. Capture the structure of PyTorch programs with minimal user intervention.
2. Use that structure to optimize and run them.

PyTorch

Models are Python programs

+ Simple
+ Debuggable
+ Hackable

- Need the Python interpreter
- Difficult to optimize

PyTorch *Eager Mode*

Models are Python programs

+ Simple
+ Debuggable
+ Hackable

- Need the Python interpreter
- Difficult to optimize

PyTorch *Eager Mode*

Models are Python programs

+ Simple
+ Debuggable
+ Hackable

- Need the Python interpreter
- Difficult to optimize

PyTorch *Script Mode*

Models are written in a high-performance subset of Python

+ Optimizable
+ Production deployment
+ Compiled and run by a lean just-in-time (JIT) interpreter.

PyTorch JIT

We need a system that can:

1. Capture the structure of PyTorch programs with minimal user intervention.
2. Use that structure to optimize and run them.

Tools to transition from Eager to Script

EAGER MODE

For prototyping, training, experimenting

`@torch.jit.script`

`torch.jit.trace()`

SCRIPT MODE

For production deployment

Eager to Script mode with `torch.jit.trace()`

Take an existing eager model, and provide example inputs.

The tracer runs the function, recording the tensor operations performed.

We turn the recording into a TorchScript module.

- Can reuse existing eager model code
- ⚠ Control-flow is ignored

```
import torch
import torchvision

def foo(x, y):
  return 2 * x + y

# trace a model by providing example inputs
traced_foo = torch.jit.trace(foo,
                             (torch.rand(3), torch.rand(3)))

traced_resnet = torch.jit.trace(torchvision.models.resnet18(),
                                torch.rand(1, 3, 224, 224))
```

Eager to Script mode with `@torch.jit.script`

Write model directly in TorchScript, a high-performance subset of Python.

Annotate with **`@torch.jit.script`** or **`@torch.jit.script_method`**

- Control-flow is preserved
- **`print`** statements can be used for debugging
- Remove the annotations to debug using standard Python tools.

```
class RNN(torch.jit.ScriptModule):
  def __init__(self, W_h, U_h, W_y, b_h, b_y):
    super(RNN, self).__init__()
    self.W_h = nn.Parameter(W_h)
    self.U_h = nn.Parameter(U_h)
    self.W_y = nn.Parameter(W_y)
    self.b_h = nn.Parameter(b_h)
    self.b_y = nn.Parameter(b_y)

  @torch.jit.script_method
  def forward(self, x, h):
    y = []
    for t in range(x.size(0)):
      h = torch.tanh(x[t] @ self.W_h + h @ self.U_h + self.b_h)
      y += [torch.tanh(h @ self.W_y + self.b_y)]
      if t % 10 == 0:
        print("stats: ", h.mean(), h.var())
    return torch.stack(y), h
```

Exporting a model to production

TorchScript models can be saved a model archive, and loaded to run in PyTorch's just-in-time (JIT) compiler instead of the CPython interpreter.

C++ Tensor APIs support bindings to a wide range of languages and deployment environments.

```python
# Python: save model
traced_resnet = torch.jit.trace(torchvision.models.resnet18(),
                                torch.rand(1, 3, 224, 224))
traced_resnet.save("serialized_resnet.pt")
```

```cpp
// C++: load model
auto module = torch::jit::load("serialized_resnet.pt");
auto example = torch::rand({1, 3, 224, 224});

// Execute `forward()` using the PyTorch JIT
auto output = module->forward({example}).toTensor();
std::cout << output.slice(1, 0, 5) << '\n';
```

PyTorch JIT

We need a system that can:

1. Capture the structure of PyTorch programs with minimal user intervention.
2. Use that structure to optimize and run them.

PyTorch JIT Intermediate Representation

```
graph(%0 : Float(3, 10), %1 : Float(3, 20), %2 : Float(3, 20), %3 : Float(80, 10)
      %4 : Float(80, 20), %5 : Float(80), %6 : Float(80)) {
  %7 : Float(10!, 80!) = aten::t(%3)
  %10 : int[] = prim::ListConstruct(3, 80)
  %12 : Float(3!, 80) = aten::expand(%5, %10, 1)
  %15 : Float(3, 80) = aten::addmm(%12, %0, %7, 1, 1)
  %16 : Float(20!, 80!) = aten::t(%4)
  %19 : int[] = prim::ListConstruct(3, 80)
  %21 : Float(3!, 80) = aten::expand(%6, %19, True)
  %24 : Float(3, 80) = aten::addmm(%21, %1, %16, 1, 1)
  %26 : Float(3, 80) = aten::add(%15, %24, 1)
  %29 : Dynamic[] = aten::chunk(%26, 4, 1)
  %30 : Float(3!, 20), %31 : Float(3!, 20), %32 : Float(3!, 20), %33 : Float(3!, 20) =
prim::ListUnpack(%29)
  %34 : Float(3, 20) = aten::sigmoid(%30)
  %35 : Float(3, 20) = aten::sigmoid(%31)
  %36 : Float(3, 20) = aten::tanh(%32)
  %37 : Float(3, 20) = aten::sigmoid(%33)
  %38 : Float(3, 20) = aten::mul(%35, %2)
  %39 : Float(3, 20) = aten::mul(%34, %36)
  %41 : Float(3, 20) = aten::add(%38, %39, 1)
  %42 : Float(3, 20) = aten::tanh(%41)
  %43 : Float(3, 20) = aten::mul(%37, %42)
  return (%43, %41);
}
```

PyTorch JIT Intermediate Representation

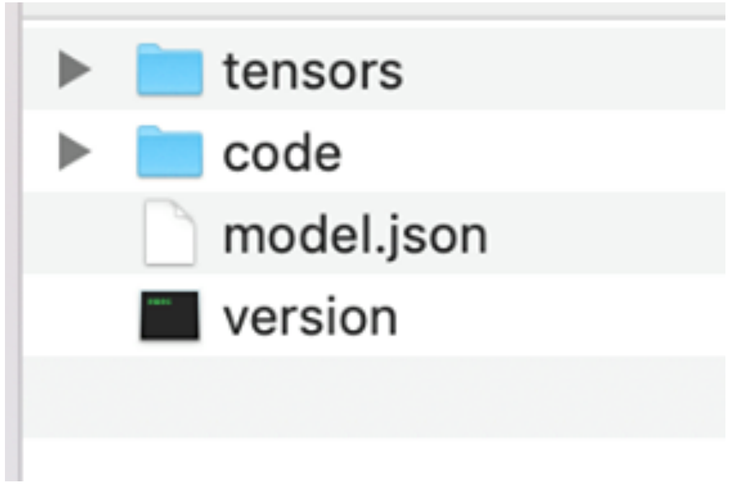
```
graph(%0 : Float(3, 10), %1 : Float(3, 20), %2 : Float(3, 20), %3 : Float(80, 10)
      %4 : Float(80, 20), %5 : Float(80), %6 : Float(80)) {
  %7 : Float(10!, 80!) = aten::t(%3)
  %10 : int[] = prim::ListConstruct(3, 80)
  %12 : Float(3!, 80) = aten::expand(%5, %10, 1)
  %15 : Float(3, 80) = aten::addmm(%12, %0, %7, 1, 1)
  %16 : Float(20!, 80!) = aten::t(%4)
  %19 : int[] = prim::Lis
  %21 : Float(3!, 80) = a
  %24 : Float(3, 80) = a
  %26 : Float(3, 80) = a
  %29 : Dynamic[] = aten
  %30 : Float(3!, 20), %31 : Float(3!, 20), %32 : Float(3!, 20), %33 : Float(3!, 20) =
prim::ListUnpack(%29)
  %34 : Float(3, 20) = aten::sigmoid(%30)
  %35 : Float(3, 20) = aten::sigmoid(%31)
  %36 : Float(3, 20) = aten::tanh(%32)
  %37 : Float(3, 20) = aten::sigmoid(%33)
  %38 : Float(3, 20) = aten::mul(%35, %2)
  %39 : Float(3, 20) = aten::mul(%34, %36)
  %41 : Float(3, 20) = aten::add(%38, %39, 1)
  %42 : Float(3, 20) = aten::tanh(%41)
  %43 : Float(3, 20) = aten::mul(%37, %42)
  return (%43, %41);
}
```

- Static typing
- Structured control flow (nested ifs and loops)
- Functional by default

PyTorch JIT Serialization

```
model = torch.jit.trace(torchvision.models.resnet18(), torch.randn(1, 3, 224, 224))
model.save('resnet.zip')
```

PyTorch JIT Serialization



PyTorch JIT Serialization

0	Today at 3:14 PM	8 bytes
1	Today at 3:14 PM	38 KB
2	Today at 3:14 PM	256 bytes
3	Today at 3:14 PM	256 bytes
4	Today at 3:14 PM	256 bytes
5	Today at 3:14 PM	256 bytes
6	Today at 3:14 PM	8 bytes
7	Today at 3:14 PM	147 KB
8	Today at 3:14 PM	256 bytes
9	Today at 3:14 PM	256 bytes
10	Today at 3:14 PM	256 bytes
11	Today at 3:14 PM	256 bytes
12	Today at 3:14 PM	8 bytes
13	Today at 3:14 PM	147 KB
14	Today at 3:14 PM	256 bytes
15	Today at 3:14 PM	256 bytes
16	Today at 3:14 PM	256 bytes
17	Today at 3:14 PM	256 bytes
18	Today at 3:14 PM	8 bytes
19	Today at 3:14 PM	147 KB
20	Today at 3:14 PM	256 bytes
21	Today at 3:14 PM	256 bytes
22	Today at 3:14 PM	256 bytes
23	Today at 3:14 PM	256 bytes
24	Today at 3:14 PM	8 bytes
25	Today at 3:14 PM	147 KB
26	Today at 3:14 PM	256 bytes
27	Today at 3:14 PM	256 bytes
28	Today at 3:14 PM	256 bytes
29	Today at 3:14 PM	256 bytes
30	Today at 3:14 PM	8 bytes
31	Today at 3:14 PM	295 KB
32	Today at 3:14 PM	512 bytes
33	Today at 3:14 PM	512 bytes
34	Today at 3:14 PM	512 bytes
35	Today at 3:14 PM	512 bytes
36	Today at 3:14 PM	8 bytes
37	Today at 3:14 PM	590 KB
38	Today at 3:14 PM	512 bytes
39	Today at 3:14 PM	512 bytes
40	Today at 3:14 PM	512 bytes
41	Today at 3:14 PM	512 bytes
42	Today at 3:14 PM	8 bytes
43	Today at 3:14 PM	33 KB
44	Today at 3:14 PM	512 bytes

PyTorch JIT Serialization

```
op_version_set = 0
def forward(self,
    input_1: Tensor) -> Tensor:
  input_2 = torch._convolution(input_1, self.conv1.weight, None, [2, 2], [3, 3], [1, 1], False, [0, 0], 1, False, False, True)
  _0 = torch.add_(self.bn1.num_batches_tracked, CONSTANTS.c0, alpha=1)
  input_3 = torch.batch_norm(input_2, self.bn1.weight, self.bn1.bias, self.bn1.running_mean, self.bn1.running_var, True, 0.10000000000000001, 1.0000000000000001e-05, True)
  input_4 = torch.threshold_(input_3, 0., 0.)
  input_5, _1 = torch.max_pool2d_with_indices(input_4, [3, 3], [2, 2], [1, 1], [1, 1], False)
  input_6 = torch._convolution(input_5, getattr(self.layer1, "0").conv1.weight, None, [1, 1], [1, 1], [1, 1], False, [0, 0], 1, False, False, True)
  _2 = torch.add_(getattr(self.layer1, "0").bn1.num_batches_tracked, CONSTANTS.c0, alpha=1)
  input_7 = torch.batch_norm(input_6, getattr(self.layer1, "0").bn1.weight, getattr(self.layer1, "0").bn1.bias, getattr(self.layer1, "0").bn1.running_mean, getattr(self.layer1, "0").bn1.running_var, True, 0.10000000000000001, 1.0000000000000001e-05, True)
  input_8 = torch.threshold_(input_7, 0., 0.)
  input_9 = torch._convolution(input_8, getattr(self.layer1, "0").conv2.weight, None, [1, 1], [1, 1], [1, 1], False, [0, 0], 1, False, False, True)
  _3 = torch.add_(getattr(self.layer1, "0").bn2.num_batches_tracked, CONSTANTS.c0, alpha=1)
  out_1 = torch.batch_norm(input_9, getattr(self.layer1, "0").bn2.weight, getattr(self.layer1, "0").bn2.bias, getattr(self.layer1, "0").bn2.running_mean, getattr(self.layer1, "0").bn2.running_var, True, 0.10000000000000001, 1.0000000000000001e-05, True)
  input_10 = torch.add_(out_1, input_5, alpha=1)
  input_11 = torch.threshold_(input_10, 0., 0.)
  input_12 = torch._convolution(input_11, getattr(self.layer1, "1").conv1.weight, None, [1, 1], [1, 1], [1, 1], False, [0, 0], 1, False, False, True)
  _4 = torch.add_(getattr(self.layer1, "1").bn1.num_batches_tracked, CONSTANTS.c0, alpha=1)
  input_13 = torch.batch_norm(input_12, getattr(self.layer1, "1").bn1.weight, getattr(self.layer1, "1").bn1.bias, getattr(self.layer1, "1").bn1.running_mean, getattr(self.layer1, "1").bn1.running_var, True, 0.10000000000000001, 1.0000000000000001e-05, True)
  input_14 = torch.threshold_(input_13, 0., 0.)
  input_15 = torch._convolution(input_14, getattr(self.layer1, "1").conv2.weight, None, [1, 1], [1, 1], [1, 1], False, [0, 0], 1, False, False, True)
  _5 = torch.add_(getattr(self.layer1, "1").bn2.num_batches_tracked, CONSTANTS.c0, alpha=1)
  out_2 = torch.batch_norm(input_15, getattr(self.layer1, "1").bn2.weight, getattr(self.layer1, "1").bn2.bias, getattr(self.layer1, "1").bn2.running_mean, getattr(self.layer1, "1").bn2.running_var, True, 0.10000000000000001, 1.0000000000000001e-05, True)
  input_16 = torch.add_(out_2, input_11, alpha=1)
  input_17 = torch.threshold_(input_16, 0., 0.)
  input_18 = torch._convolution(input_17, getattr(self.layer2, "0").conv1.weight, None, [2, 2], [1, 1], [1, 1], False, [0, 0], 1, False, False, True)
  _6 = torch.add_(getattr(self.layer2, "0").bn1.num_batches_tracked, CONSTANTS.c0, alpha=1)
  input_19 = torch.batch_norm(input_18, getattr(self.layer2, "0").bn1.weight, getattr(self.layer2, "0").bn1.bias, getattr(self.layer2, "0").bn1.running_mean, getattr(self.layer2, "0").bn1.running_var, True, 0.10000000000000001, 1.0000000000000001e-05, True)
  input_20 = torch.threshold_(input_19, 0., 0.)
  input_21 = torch._convolution(input_20, getattr(self.layer2, "0").conv2.weight, None, [1, 1], [1, 1], [1, 1], False, [0, 0], 1, False, False, True)
  _7 = torch.add_(getattr(self.layer2, "0").bn2.num_batches_tracked, CONSTANTS.c0, alpha=1)
  out_3 = torch.batch_norm(input_21, getattr(self.layer2, "0").bn2.weight, getattr(self.layer2, "0").bn2.bias, getattr(self.layer2, "0").bn2.running_mean, getattr(self.layer2, "0").bn2.running_var, True, 0.10000000000000001, 1.0000000000000001e-05, True)
  input_22 = torch._convolution(input_17, getattr(getattr(self.layer2, "0").downsample, "0").weight, None, [2, 2], [0, 0], [1, 1], False, [0, 0], 1, False, False, True)
  _8 = torch.add_(getattr(getattr(self.layer2, "0").downsample, "1").num_batches_tracked, CONSTANTS.c0, alpha=1)
  identity_1 = torch.batch_norm(input_22, getattr(getattr(self.layer2, "0").downsample, "1").weight, getattr(getattr(self.layer2, "0").downsample, "1").bias, getattr(getattr(self.layer2, "0").downsample, "1").running_mean, getattr(getattr(self.layer2, "0").dow
  input_23 = torch.add_(out_3, identity_1, alpha=1)
  input_24 = torch.threshold_(input_23, 0., 0.)
  input_25 = torch._convolution(input_24, getattr(self.layer2, "1").conv1.weight, None, [1, 1], [1, 1], [1, 1], False, [0, 0], 1, False, False, True)
  _9 = torch.add_(getattr(self.layer2, "1").bn1.num_batches_tracked, CONSTANTS.c0, alpha=1)
  input_26 = torch.batch_norm(input_25, getattr(self.layer2, "1").bn1.weight, getattr(self.layer2, "1").bn1.bias, getattr(self.layer2, "1").bn1.running_mean, getattr(self.layer2, "1").bn1.running_var, True, 0.10000000000000001, 1.0000000000000001e-05, True)
  input_27 = torch.threshold_(input_26, 0., 0.)
  input_28 = torch._convolution(input_27, getattr(self.layer2, "1").conv2.weight, None, [1, 1], [1, 1], [1, 1], False, [0, 0], 1, False, False, True)
  _10 = torch.add_(getattr(self.layer2, "1").bn2.num_batches_tracked, CONSTANTS.c0, alpha=1)
  out_4 = torch.batch_norm(input_28, getattr(self.layer2, "1").bn2.weight, getattr(self.layer2, "1").bn2.bias, getattr(self.layer2, "1").bn2.running_mean, getattr(self.layer2, "1").bn2.running_var, True, 0.10000000000000001, 1.0000000000000001e-05, True)
  input_29 = torch.add_(out_4, input_24, alpha=1)
  input_30 = torch.threshold_(input_29, 0., 0.)
  input_31 = torch._convolution(input_30, getattr(self.layer3, "0").conv1.weight, None, [2, 2], [1, 1], [1, 1], False, [0, 0], 1, False, False, True)
  _11 = torch.add_(getattr(self.layer3, "0").bn1.num_batches_tracked, CONSTANTS.c0, alpha=1)
```

What kind of optimizations do we want to do?

Algebraic rewriting

Constant folding, common subexpression elimination, dead code elimination, etc.

Out-of-order execution

Re-ordering operations to reduce memory pressure and make efficient use of cache locality

Kernel fusion

Combining several operators into a single kernel to avoid per-op overhead

Target-dependent code generation

Taking parts of the program and compiling them for specific hardware.

Integration ongoing with several codegen frameworks: TVM, Halide, Glow, XLA

Maintaining the same semantics is critical for user experience

Users should get optimization "for free"!

Dynamism in PyTorch Script Mode

Even with TorchScript's more static semantics, we want to preserve the flexibility and ease of use of Eager mode.

That means there's still a lot of dynamism in TorchScript programs!

Dynamism in PyTorch Script Mode

```
@torch.jit.script
def lstm(self, input, hidden):
    hx, cx = hidden
    gates = (F.linear(input, self.w_ih, self.b_ih) + \
             F.linear(hx, self.w_hh, self.b_hh))
    ingate, forgetgate, cellgate, outgate = \
      gates.chunk(4, 1)
    ingate = F.sigmoid(ingate)
    forgetgate = F.sigmoid(forgetgate)
    cellgate = F.tanh(cellgate)
    outgate = F.sigmoid(outgate)
    cy = (forgetgate * cx) + (ingate * cellgate)
    hy = outgate * F.tanh(cy)
    return hy, cy
```

Can we fuse this LSTM cell?

- What devices are the tensors on?
- How many dimensions?
- How big are those dimensions?
- Are we using autograd?

Optimization via JIT Compilation

In production environments, input properties are likely to remain static.

Whenever you have something that *can be* dynamic, but *is likely* static, just-in-time compilation may be useful!

Optimization via JIT Compilation

User calls `forward()`

Specialization

- Shape
- Device
- requires_grad

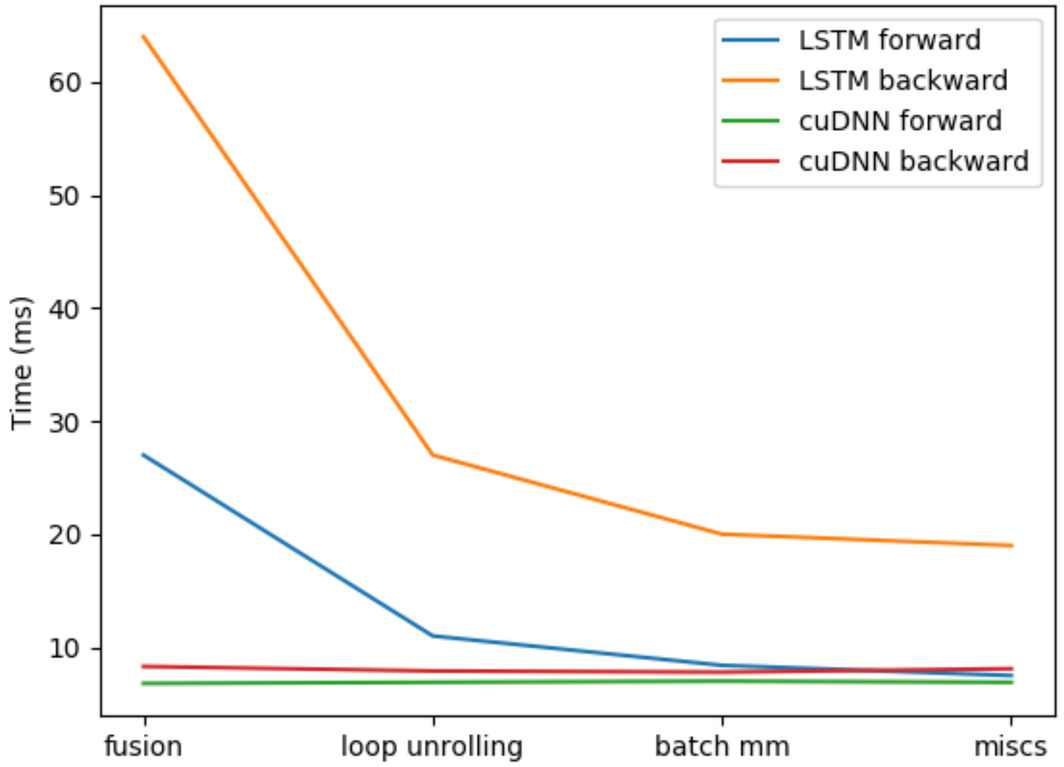
Compiler optimizations

- Graph fusion
- Algebraic rewriting
- Loop unrolling
- Code generation
- etc...

Execution

- Scheduling
- Parallelism

Do models actually get faster?



Speeding-up PyTorch with JIT

- Where does the speed-up comes from?
  - Fusing operations into a single kernel
  - Use `.graph_for` in a torchscript function to get the graph out of it

Speeding-up PyTorch with JIT

- Where does the speed-up comes from?

  - Fusing operations into a single kernel

  - Use `.graph_for` in a torchscript function to get the graph out of it

```
import torch

# necessary for CPU fusion for now
torch._C._jit_override_can_fuse_on_cpu(True)

@torch.jit.script
def script_fn(x):
    for i in range(5):
        x = x * x
    return x

a = torch.rand(5)
script_fn(a)
print(script_fn.graph_for(a))
```

Speeding-up PyTorch with JIT

- Where does the speed-up comes from?

  - Fusing operations into a single kernel

  - Use `.graph_for` in a torchscript function to get the graph out of it

```
import torch

# necessary for CPU fusion for now
torch._C._jit_override_can_fuse_on_cpu(True)

@torch.jit.script
def script_fn(x):
    for i in range(5):
        x = x * x
    return x

a = torch.rand(5)
script_fn(a)
print(script_fn.graph_for(a))
```

```
graph(%x.1 : Float(*)):
  %x.5 : Float(*) = prim::FusionGroup_0(%x.1)
  return (%x.5)
with prim::FusionGroup_0 = graph(%8 : Float(*)):
  %x.6 : Float(*) = aten::mul(%8, %8)
  %x.2 : Float(*) = aten::mul(%x.6, %x.6)
  %x.3 : Float(*) = aten::mul(%x.2, %x.2)
  %x.4 : Float(*) = aten::mul(%x.3, %x.3)
  %x.5 : Float(*) = aten::mul(%x.4, %x.4)
  return (%x.5)
```

Speeding-up PyTorch with JIT

- Where does the speed-up comes from?

  - Fusing operations into a single kernel

  - Use `.graph_for` in a torchscript function to get the graph out of it

```
import torch

# necessary for CPU fusion for now
torch._C._jit_override_can_fuse_on_cpu(True)

@torch.jit.script
def script_fn(x):
    for i in range(5):
        x = x * x
    return x

a = torch.rand(5)
script_fn(a)
print(script_fn.graph_for(a))
```

```
graph(%x.1 : Float(*)):
  %x.5 : Float(*) = prim::FusionGroup_0(%x.1)
  return (%x.5)
with prim::FusionGroup_0 = graph(%8 : Float(*)):
  %x.6 : Float(*) = aten::mul(%8, %8)
  %x.2 : Float(*) = aten::mul(%x.6, %x.6)
  %x.3 : Float(*) = aten::mul(%x.2, %x.2)
  %x.4 : Float(*) = aten::mul(%x.3, %x.3)
  %x.5 : Float(*) = aten::mul(%x.4, %x.4)
  return (%x.5)
```

Speeding-up PyTorch with JIT

- Where does the speed-up comes from?

  - Fusing operations into a single kernel

  - Use `.graph_for` in a torchscript function to get the graph out of it

```
import torch

# necessary for CPU fusion for now
torch._C._jit_override_can_fuse_on_cpu(True)

@torch.jit.script
def script_fn(x):
    for i in range(5):
        x = x * x
    return x

a = torch.rand(5)
script_fn(a)
print(script_fn.graph_for(a))
```

```
graph(%x.1 : Float(*)):
  %x.5 : Float(*) = prim::FusionGroup_0(%x.1)
  return (%x.5)
with prim::FusionGroup_0 = graph(%8 : Float(*)):
  %x.6 : Float(*) = aten::mul(%8, %8)
  %x.2 : Float(*) = aten::mul(%x.6, %x.6)
  %x.3 : Float(*) = aten::mul(%x.2, %x.2)
  %x.4 : Float(*) = aten::mul(%x.3, %x.3)
  %x.5 : Float(*) = aten::mul(%x.4, %x.4)
  return (%x.5)
```

Speeding-up PyTorch with JIT

- Where does the speed-up comes from?

  - Fusing operations into a single kernel

  - Use `.graph_for` in a torchscript function to get the graph out of it

```
import torch

# necessary for CPU fusion for now
torch._C._jit_override_can_fuse_on_cpu(True)

@torch.jit.script
def script_fn(x):
    for i in range(5):
        x = x * x
    return x

a = torch.rand(5)
script_fn(a)
print(script_fn.graph_for(a))
```

```
graph(%x.1 : Float(*)):
  %x.5 : Float(*) = prim::FusionGroup_0(%x.1)
  return (%x.5)
with prim::FusionGroup_0 = graph(%8 : Float(*)):
  %x.6 : Float(*) = aten::mul(%8, %8)
  %x.2 : Float(*) = aten::mul(%x.6, %x.6)
  %x.3 : Float(*) = aten::mul(%x.2, %x.2)
  %x.4 : Float(*) = aten::mul(%x.3, %x.3)
  %x.5 : Float(*) = aten::mul(%x.4, %x.4)
  return (%x.5)
```

Single kernel!

Speeding-up PyTorch with JIT

- Where does the speed-up comes from?

  - Fusing operations into a single kernel

  - Use `.graph_for` in a torchscript function to get the graph out of it

```
import torch

# necessary for CPU fusion for now
torch._C._jit_override_can_fuse_on_cpu(True)

@torch.jit.script
def script_fn(x):
    for i in range(5):
        x = x * x
    return x

a = torch.rand(5)
script_fn(a)
print(script_fn.graph_for(a))
```

```
graph(%x.1 : Float(*)):
  %x.5 : Float(*) = prim::FusionGroup_0(%x.1)
  return (%x.5)
with prim::FusionGroup_0 = graph(%8 : Float(*)):
  %x.6 : Float(*) = aten::mul(%8, %8)
  %x.2 : Float(*) = aten::mul(%x.6, %x.6)
  %x.3 : Float(*) = aten::mul(%x.2, %x.2)
  %x.4 : Float(*) = aten::mul(%x.3, %x.3)
  %x.5 : Float(*) = aten::mul(%x.4, %x.4)
  return (%x.5)
```

```
%timeit script_fn(a)
4.02 ms ± 62.5 µs per loop (mean ± std.
dev. of 7 runs, 100 loops each)
```

```
%timeit my_fn(a)
7.29 ms ± 50.1 µs per loop (mean ± std.
dev. of 7 runs, 100 loops each)
```

PyTorch JIT

A compiler infrastructure that enables gradual transition from research to production, without compromising on experience.

Introduction to JIT / TorchScript:

http://bit.ly/2WiV7En

Tooling

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
- Use the most popular debugger:

Debugging

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

```
print(foo)
```

Identifying bottlenecks

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

SNAKEVIZ

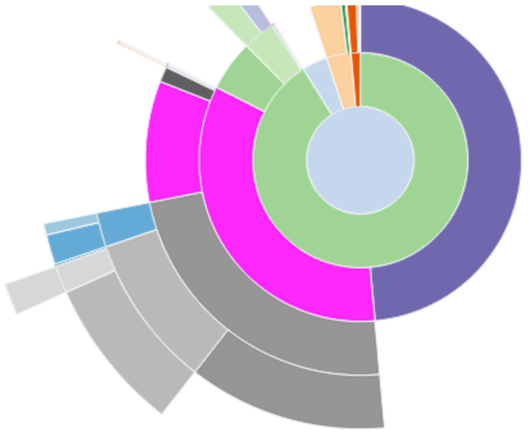
← PREVIOUS NEXT → G

SnakeViz
Installation
Starting SnakeViz
Generating Profiles
Interpreting Results
Controls
Notes
Contact

FUNCTION INFO

Placing your cursor over an arc will highlight that arc and any other visible instances of the same function call. It also display a list of information to the left of the sunburst.

Name:
filter
Cumulative Time:
0.000294 s (31.78 %)
File:
fnmatch.py
Line:
48
Directory:
/Users/jiffyclub/miniconda3/envs/snakevizdev/lib/python3.4/



Identifying bottlenecks

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

```
File: pystone.py
Function: Proc2 at line 149
Total time: 0.606656 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   149                                           @profile
   150                                           def Proc2(IntParIO):
   151     50000        82003      1.6     13.5      IntLoc = IntParIO + 10
   152     50000        63162      1.3     10.4      while 1:
   153     50000        69065      1.4     11.4          if Char1Glob == 'A':
   154     50000        66354      1.3     10.9              IntLoc = IntLoc - 1
   155     50000        67263      1.3     11.1              IntParIO = IntLoc - IntGlob
   156     50000        65494      1.3     10.8              EnumLoc = Ident1
   157     50000        68001      1.4     11.2          if EnumLoc == Ident1:
   158     50000        63739      1.3     10.5              break
   159     50000        61575      1.2     10.1      return IntParIO
```

Identifying bottlenecks

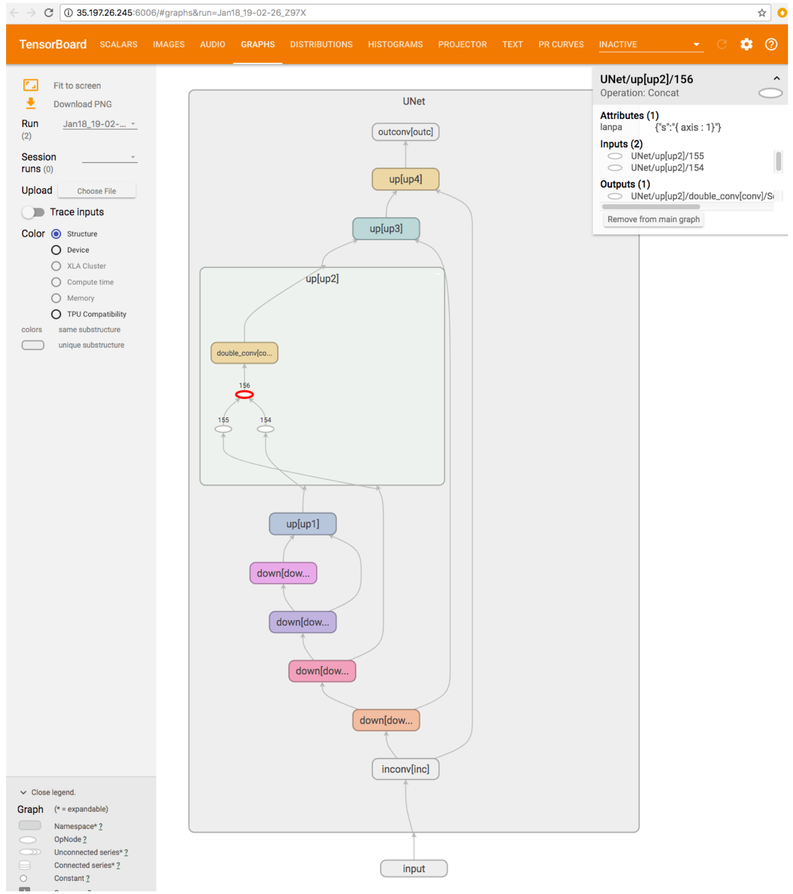
- torch.autograd.profiler

```
>>> x = torch.randn((1, 1), requires_grad=True)
>>> with torch.autograd.profiler.profile() as prof:
...     y = x ** 2
...     y.backward()
>>> # NOTE: some columns were removed for brevity
... print(prof)
-----------------------------------  --------------  --------------
Name                                        CPU time        CUDA time
-----------------------------------  --------------  --------------
PowConstant                               142.036us         0.000us
N5torch8autograd9GraphRootE                63.524us         0.000us
PowConstantBackward                       184.228us         0.000us
MulConstant                                50.288us         0.000us
PowConstant                                28.439us         0.000us
Mul                                        20.154us         0.000us
N5torch8autograd14AccumulateGradE          13.790us         0.000us
N5torch8autograd5CloneE                     4.088us         0.000us
```

Visualization

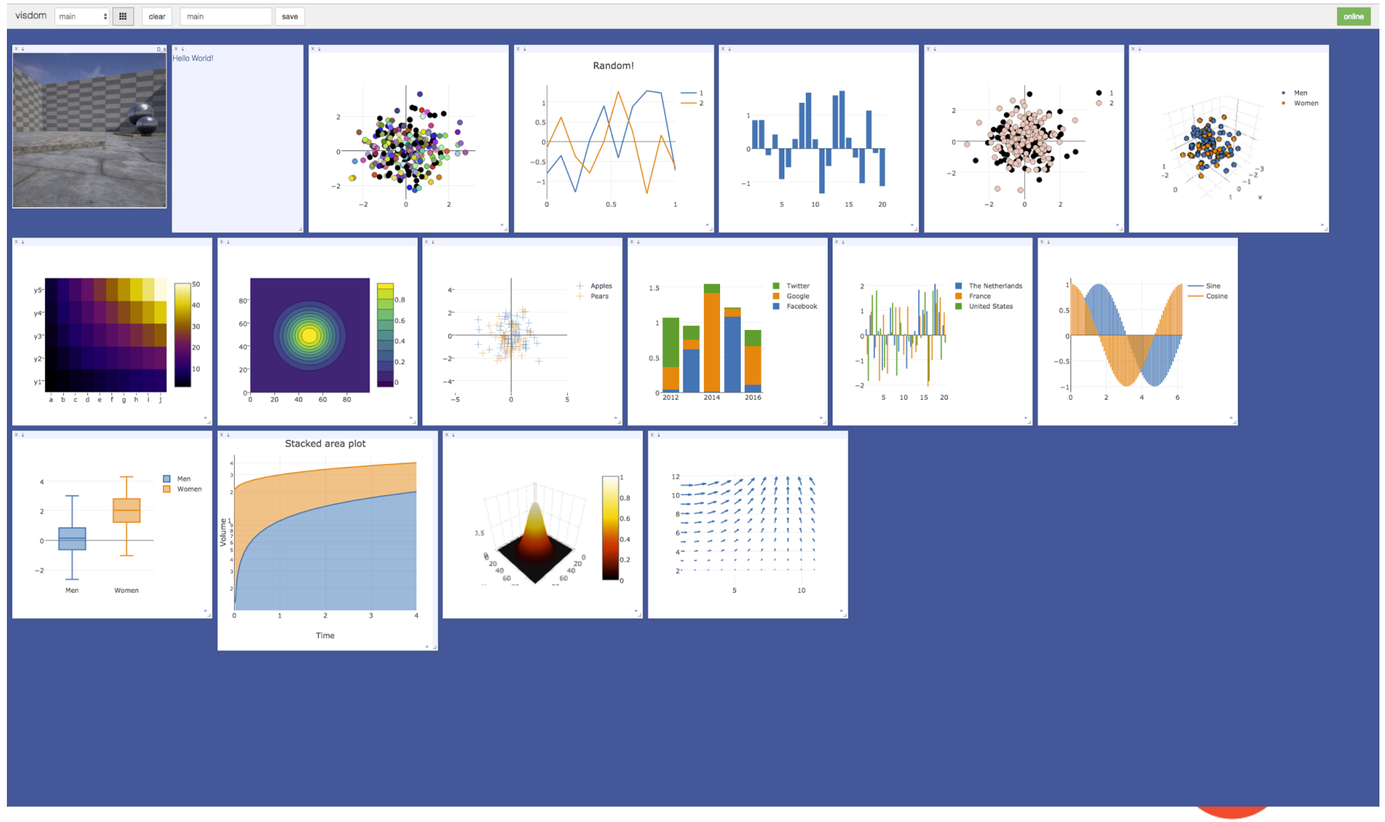
TensorBoard

torch.utils.tensorboard (as of v1.1.0)



Visdom

https://github.com/facebookresearch/visdom



Upcoming Features

RPC in torch.distributed (API not final)

```
# synchronous
result = torch.rpc(remote_function, *args, on : device)

# asynchronous
future = torch.rpc(remote_function, *args, on : device)
result = future.get()

# remote
remote_tensor = torch.remote(remote_function, *args, on : device)
```

RPC in torch.distributed (API not final)

```
# [RFC] proposed hi-perf API
# 1. load data
inputs_rref = torch.remote(load_inputs, path_to_inputs, dev1)
labels_rref = torch.remote(load_labels, path_to_inputs, dev2)

# 2. define model
class Net1(nn.Module):
    ...

class Net2(nn.Module):
    ...

class DMP(nn.Module):
    def __init__(self, dev1, dev2):
        self.net1 = torch.remote(Net1, dev1)
        self.net2 = torch.remote(Net2, dev2)

    def forward(self, inputs_rref):
        # RRef[T].__call__(args) is a sugar that translates to
        # dist.remote(T, RRef.on(), args)
        outputs1_rref = self.net1(inputs_rref)
        outputs2_rref = self.net2(outputs1_rref)
        return outputs2_rref

# 3. training, run it where you want to call autograd
def train(inputs_rref, labels_rref):
    dmp = DMP()
    # torch.optim.remote creates an optimizer on every RRef destination
    optimizer = torch.optim.remote(torch.optim.SGD, dmp, lr=0.1)
    outputs_rref = dmp(inputs_rref)
    loss = loss_func(outputs_rref.to_here(), labels_rref.to_here())
    loss.backward()
    optimize.step()

dist.rpc(train, dev2, inputs_rref, labels_rref)
```

Named Tensors

Proposal: https://goo.gl/4FDBTM

1. **Make code self-documenting.** In current PyTorch, many users add comments to their code that can easily go out of sync:

```
# Tensor[N, H, W, C]
x = torch.randn(100, 50, 50, 3)
```

Named Tensors

Proposal: https://goo.gl/4FDBTM

2. Prevent silent user logic errors through **runtime checking of names**

```
# Pointwise loss on target of size (batch) and output of size
# (batch, 1) is wrong and occurs frequently.

>>> loss = lambda x, y: (x - y)**2
>>> output = torch.randn(10, 1)
>>> targets = torch.randn(10)
>>> loss(output, targets).shape
torch.Size(10, 10)  # This is wrong
```

Named Tensors

Proposal: https://goo.gl/4FDBTM

3. Enable **better code and more expressivity** through using names in operations.

```
ims = torch.randn(100, 50, 50, 3)  # NHWC
ims2 = torch.randn(100, 3, 50, 50) # NCHW

def rotate(ims):
    return ims.transpose(1, 2)

rotate(ims)
rotate(ims2)  # NOT what we wanted
```

Named Tensors

Proposal: https://goo.gl/4FDBTM

```
>>> tensor = torch.tensor([[1, 0], [0, 1],
                           names=['height', 'width'])
You can also pass a names argument to a tensor factory function*
>>> tensor = torch.randn(2, 1, 2,
                         names=['batch', 'height', 'width'])
tensor([[[-0.9437, -1.8355]],
        [[-1.1989, -0.4061]]],
       names=['batch', 'width', 'height'])
```

Named Tensors

Proposal: https://goo.gl/4FDBTM

```
>>> tensor = torch.randn(batch=2, height=1, width=2)
```

Quantization

Proposal: https://github.com/pytorch/pytorch/issues/18318

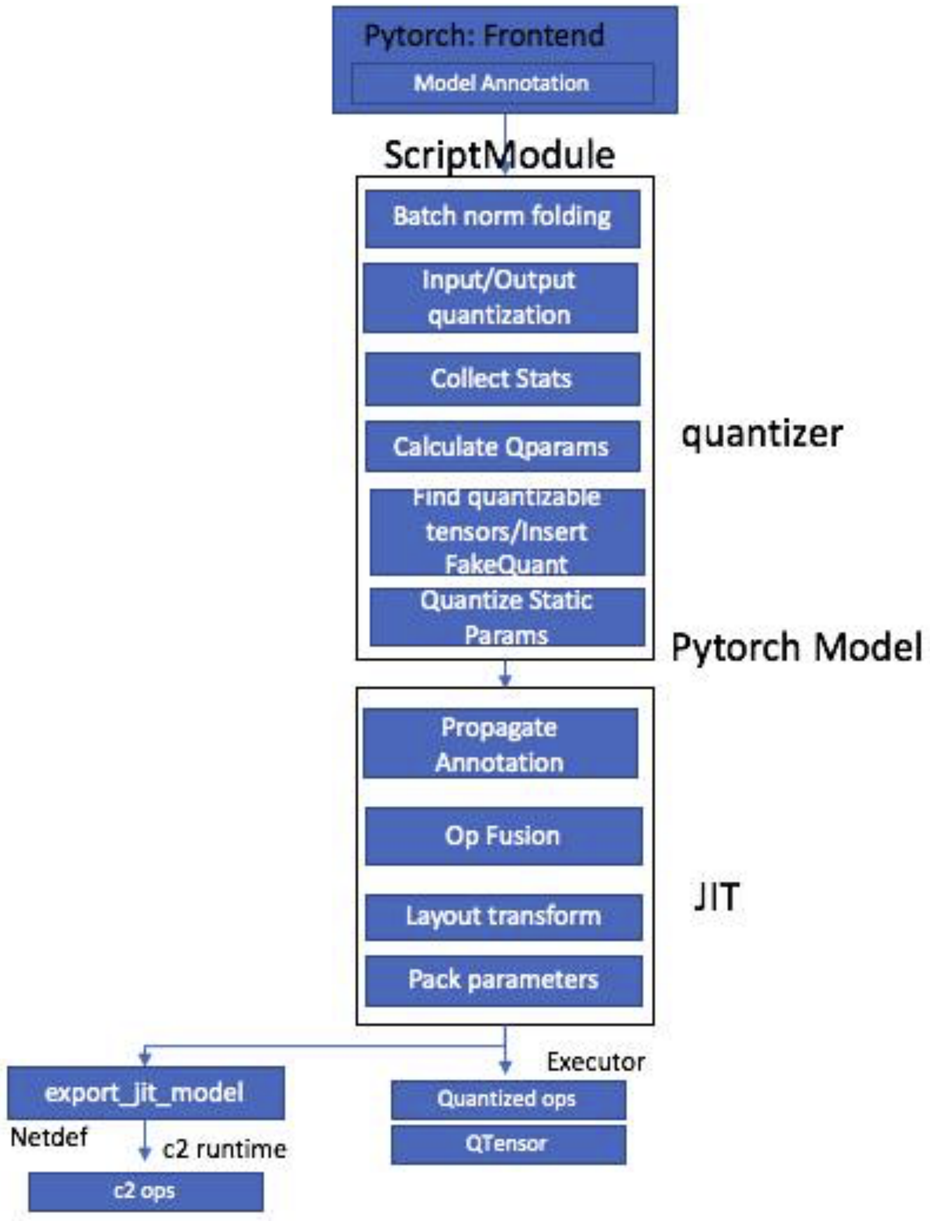
- Supported in eager and torchscript modes
- Quantize a model after training to 8-bit, 16-bit, etc.
  - Planned: Quantization-aware training
- Performance benefits

Quantization

Proposal: https://github.com/pytorch/pytorch/issues/18318

```
# Quantize to 8 bit unsigned: affine uniform quantizer
# scale is a float scalar.
# zero_point is an unsigned 8 bit integer
y = x.quantize_linear(scale, zero_point)
# y is a qtensor
# Can examine internals of y
y_scale = y.q_scale()
y_zero_point = y.q_zero_point()
print(y.int_repr())
# View quantized 8 bit values

# convert back to floating point
x_dequantized = y.dequantize()
```

Quantization



ub.com/pytorch/pytorch/issues/18318

Quantization

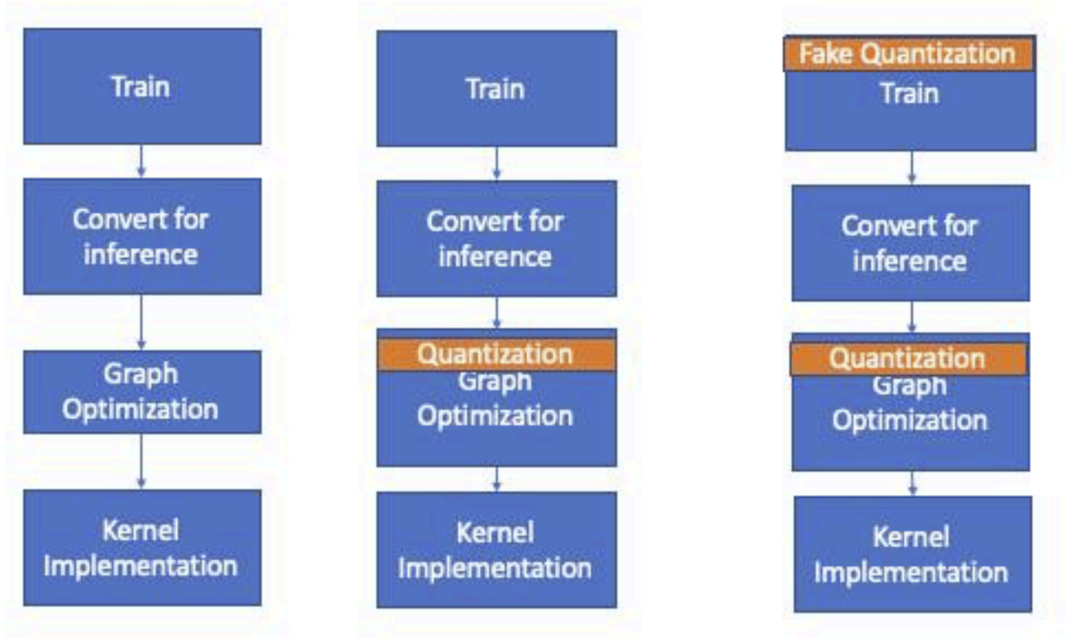
Proposal: https://github.com/pytorch/pytorch/issues/18318

```
import torch
from torch.quantization import quantizer

class Net(torch.jit.ScriptModule):
    def __init__(self):
        super(Net, self).__init__()
        self.submodule1 = baseModule((1, 20, 5, 1),qconfig1)
        self.submodule2 =baseModule((20,50,5,1),qconfig1)
        self.fc1 = nn.Linear(4*4*50, 500)
        self.fc2 = nn.Linear(500, 10)

    @torch.script_method
    def forward(x):
        ...

def eval_fn(script_module,eval_args):
...

def main():
    my_script_module = Net()
    # Minimal API: Entire model gets quantized to 8 bit precision
    qtz_script_module = quantizer.quantize(my_script_module, eval_fn, eval_args)

    qtz_script_module.save('mymodule.pt')

    # Op fusion, quantized op invocation happens under the hood at invocation
    result = qtz_script_module(test_data)
```

Quantization

Proposal: https://github.com/pytorch/pytorch/issues/18318

Model workflows, from left to right: 1. Non-quantized 2. Post-training quantization 3. Quantization-aware training



For greater accuracy, the quantization API will support modeling the effects of quanitization during training by inserting fake-quantization operators that will then be consumed by the same quantizer described above.

https://pytorch.org

With from

facebook

UBER

NVIDIA

salesforce

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIE
PARIS INSTITUTE OF TECHNOLOGY

Carnegie Mellon University

UNIVERSITE PIERRE & MARIE CURIE
LA SCIENCE A PARIS

Digital Reasoning

Stanford University

UNIVERSITY OF OXFORD

NYU

Inria

ENS
ÉCOLE NORMALE SUPÉRIEURE

EPFL
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Berkeley
UNIVERSITY OF CALIFORNIA