
PYTORCH

a framework for next-generation ML research

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Trevor Killeen, Francisco Massa, Adam Lerer, James Bradbury, Zeming Lin, Natalia Gimelshein, Christian Sarofeen, Alban Desmaison, Andreas Kopf, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, ...

But why?


TensorFlow


PYTORCH


Caffe2


dy/net


K


Chainer


mxnet

```
import tensorflow as tf
import numpy as np

N, D_in, H, D_out = 64, 1000, 100, 10

########################################################
# Step 1: Define the computational graph.
# NOTE: This is purely symbolic. Nothing is run yet

x = tf.placeholder(tf.float32, shape=(None, D_in))
y = tf.placeholder(tf.float32, shape=(None, D_out))
w1 = tf.Variable(tf.random_normal((D_in, H)))
w2 = tf.Variable(tf.random_normal((H, D_out)))

h = tf.matmul(x, w1)
h_relu = tf.maximum(h, tf.zeros(1))
y_pred = tf.matmul(h_relu, w2)
loss = tf.reduce_sum((y - y_pred) ** 2.0)

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])
new_w1 = w1.assign(w1 - 1e-6 * grad_w1)
new_w2 = w2.assign(w2 - 1e-6 * grad_w2)

########################################################
# Step 2: Fire up the VM and feed it with the data

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

    x_value = np.random.randn(N, D_in)
    y_value = np.random.randn(N, D_out)
    for _ in range(500):
        loss_value, _, _ = sess.run([loss, new_w1, new_w2],
                                    feed_dict={x: x_value, y: y_value})
        print(loss_value)  # Actual results are available only now
```

```
import tensorflow as tf
import numpy as np

N, D_in, H, D_out = 64, 1000, 100, 10

########################################################
# Step 1: Define the computational graph.
# NOTE: This is purely symbolic. Nothing is run yet

x = tf.placeholder(tf.float32, shape=(None, D_in))
y = tf.placeholder(tf.float32, shape=(None, D_out))
w1 = tf.Variable(tf.random_normal((D_in, H)))
w2 = tf.Variable(tf.random_normal((H, D_out)))

h = tf.matmul(x, w1)
h_relu = tf.maximum(h, tf.zeros(1))
y_pred = tf.matmul(h_relu, w2)
loss = tf.reduce_sum((y - y_pred) ** 2.0)

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])
new_w1 = w1.assign(w1 - 1e-6 * grad_w1)
new_w2 = w2.assign(w2 - 1e-6 * grad_w2)

########################################################
# Step 2: Fire up the VM and feed it with the data

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

    x_value = np.random.randn(N, D_in)
    y_value = np.random.randn(N, D_out)
    for _ in range(500):
        loss_value, _, _ = sess.run([loss, new_w1, new_w2],
                                    feed_dict={x: x_value, y: y_value})
        print(loss_value)  # Actual results are available only now
```

```
import tensorflow as tf
import numpy as np

N, D_in, H, D_out = 64, 1000, 100, 10

########################################################
# Step 1: Define the computational graph.
# NOTE: This is purely symbolic. Nothing is run yet

x = tf.placeholder(tf.float32, shape=(None, D_in))
y = tf.placeholder(tf.float32, shape=(None, D_out))
w1 = tf.Variable(tf.random_normal((D_in, H)))
w2 = tf.Variable(tf.random_normal((H, D_out)))

h = tf.matmul(x, w1)
h_relu = tf.maximum(h, tf.zeros(1))
y_pred = tf.matmul(h_relu, w2)
loss = tf.reduce_sum((y - y_pred) ** 2.0)

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])
new_w1 = w1.assign(w1 - 1e-6 * grad_w1)
new_w2 = w2.assign(w2 - 1e-6 * grad_w2)

########################################################
# Step 2: Fire up the VM and feed it with the data

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

    x_value = np.random.randn(N, D_in)
    y_value = np.random.randn(N, D_out)
    for _ in range(500):
        loss_value, _, _ = sess.run([loss, new_w1, new_w2],
                                    feed_dict={x: x_value, y: y_value})
        print(loss_value)  # Actual results are available only now
```

Declarative/symbolic

Custom virtual machines

Entire computation declared upfront

Easy to serialize

Easy to optimize

~~Declarative/symbolic~~

~~Custom virtual machines~~

~~Entire computation declared upfront~~

Easy to serialize

Easy to optimize

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10

##############################################################
# Step 1: Run.

w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    x = torch.randn(N, D_in)
    y = torch.randn(N, D_out)

    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
    w1.grad.data.zero_()
    w2.grad.data.zero_()
```

Minimize mental overhead

Leverage Python ecosystem

Allow free-form models

Keep things very fast

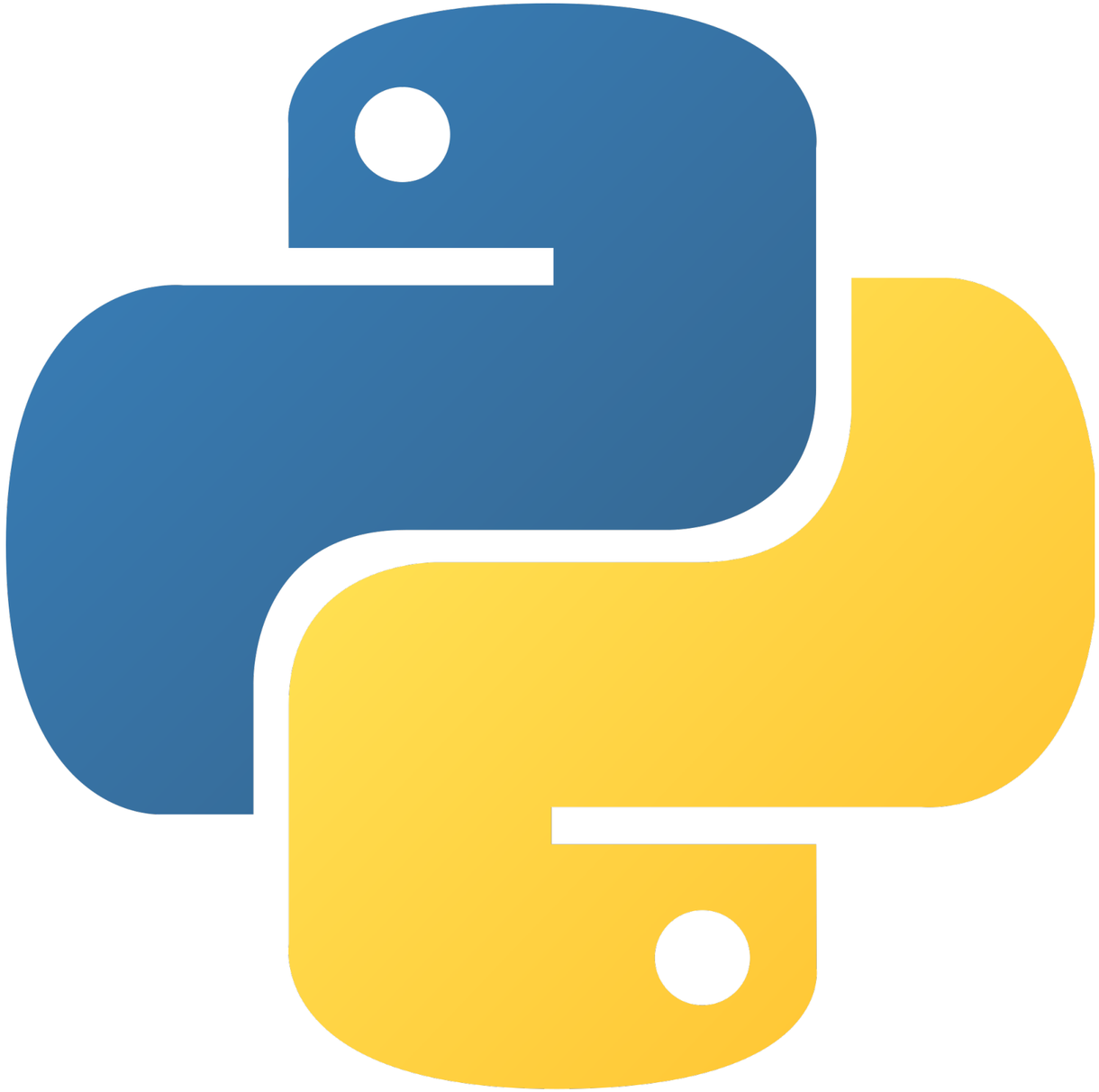
NdArray library

```
import torch

x = torch.empty(5, 5)        # Tensor creation

y = x + x                    # Regular Python operators are supported

s = x[..., -1:]              # Full basic + a large part of NumPy advanced indexing

b = x + x[:, -1:]            # Broadcasting (operands are of different shapes)

z = x.sigmoid()              # A ton of math functions as methods (+ chaining!)

q = torch.abs(x)             # Functional syntax

x.uniform_()                 # In-place modification

i = x.long()                 # Type casting

print(x)                     # Pretty-printing
```

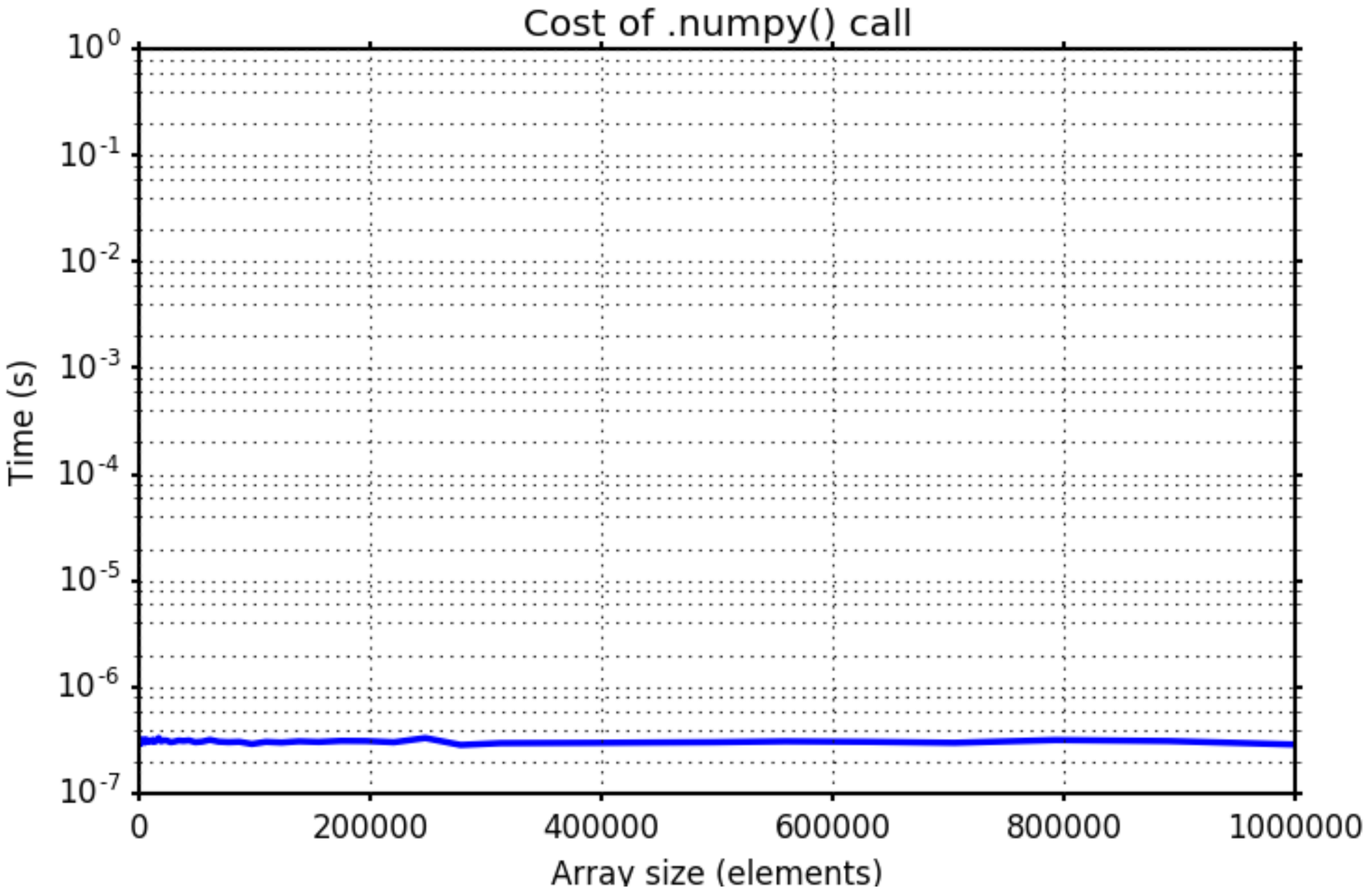

GTX 1080Ti
NVIDIA

```
# Move tensor to current GPU (0 by defualt)
c = x.cuda()

# Do some math as if everything happened on CPU
r = torch.matmul(c, c)
q = c ** 2 + r * c

# See the results (or use .cpu() to transfer back to host)
print(q)

# tensor([[-2.7569,  1.6615, -0.4716, -0.0707, -0.1471],
#         [ 0.4811,  0.3971,  1.1509,  0.1125, -0.1877],
#         [ 0.4440,  0.8234, -0.2956,  0.3831,  0.0108],
#         [ 0.5591,  0.8117, -0.2884,  2.3547,  0.9185],
#         [-0.4594,  0.4424, -0.3554,  2.1240,  1.6757]],
#         device='cuda:0')
```

```
# Manually specify device (if already on CUDA, switch devs)
c = x.cuda(1)
print(c.get_device())       # 1


# Change the default device for a block of code
with torch.cuda.device(1):
    c = x.cuda()            # no device == current device
    print(c.get_device())   # 1


# NOTE: still can do math in context of GPU 0
d = c * 2 + 1
print(d.get_device())       # 1
```



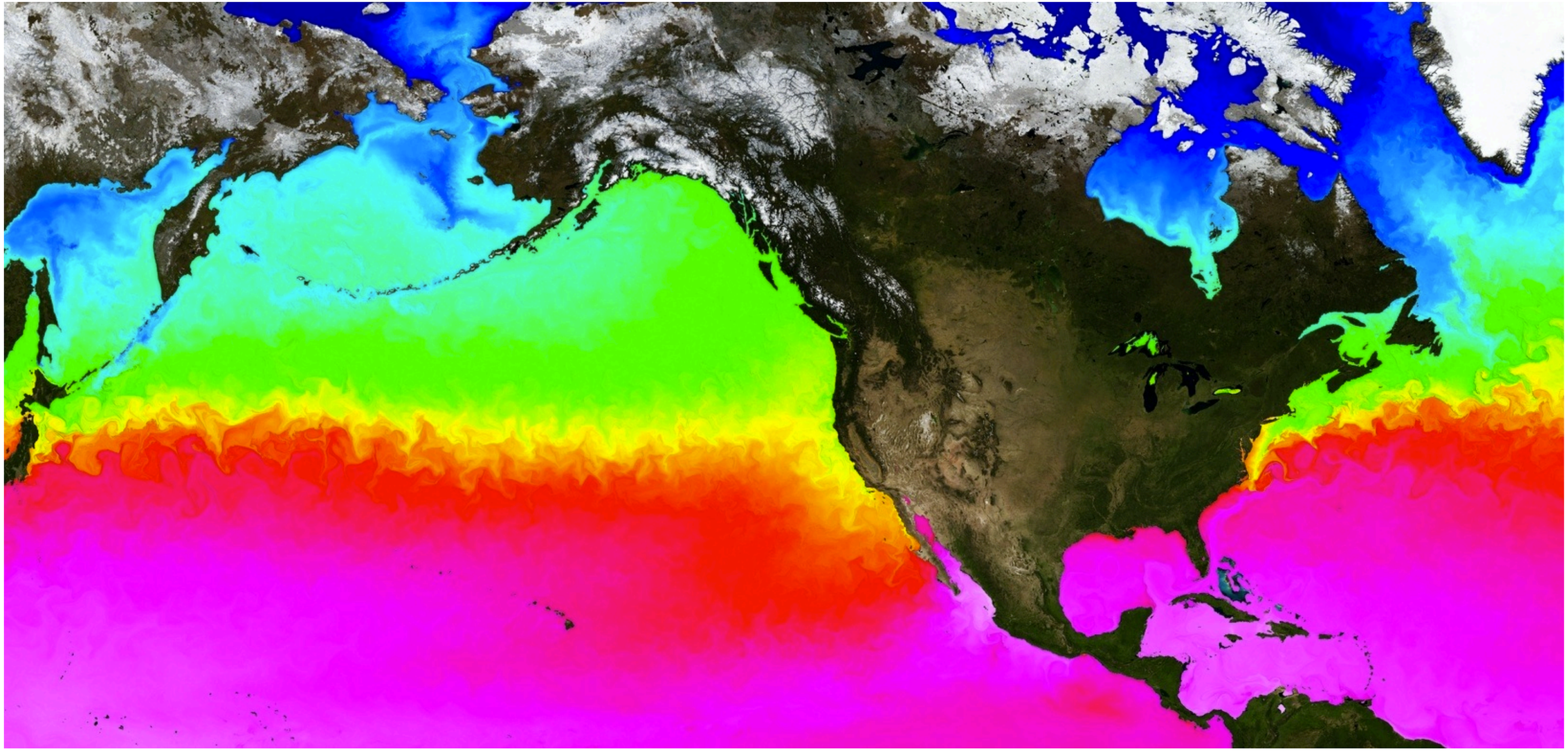
```
x = torch.ones(5, 5)
print(x)

# tensor([[1., 1., 1., 1., 1.],
#         [1., 1., 1., 1., 1.],
#         [1., 1., 1., 1., 1.],
#         [1., 1., 1., 1., 1.],
#         [1., 1., 1., 1., 1.]])

arr = x.numpy()
print(arr)

# array([[ 1.,  1.,  1.,  1.,  1.],
#        [ 1.,  1.,  1.,  1.,  1.],
#        [ 1.,  1.,  1.,  1.,  1.],
#        [ 1.,  1.,  1.,  1.,  1.],
#        [ 1.,  1.,  1.,  1.,  1.]], dtype=float32)
```

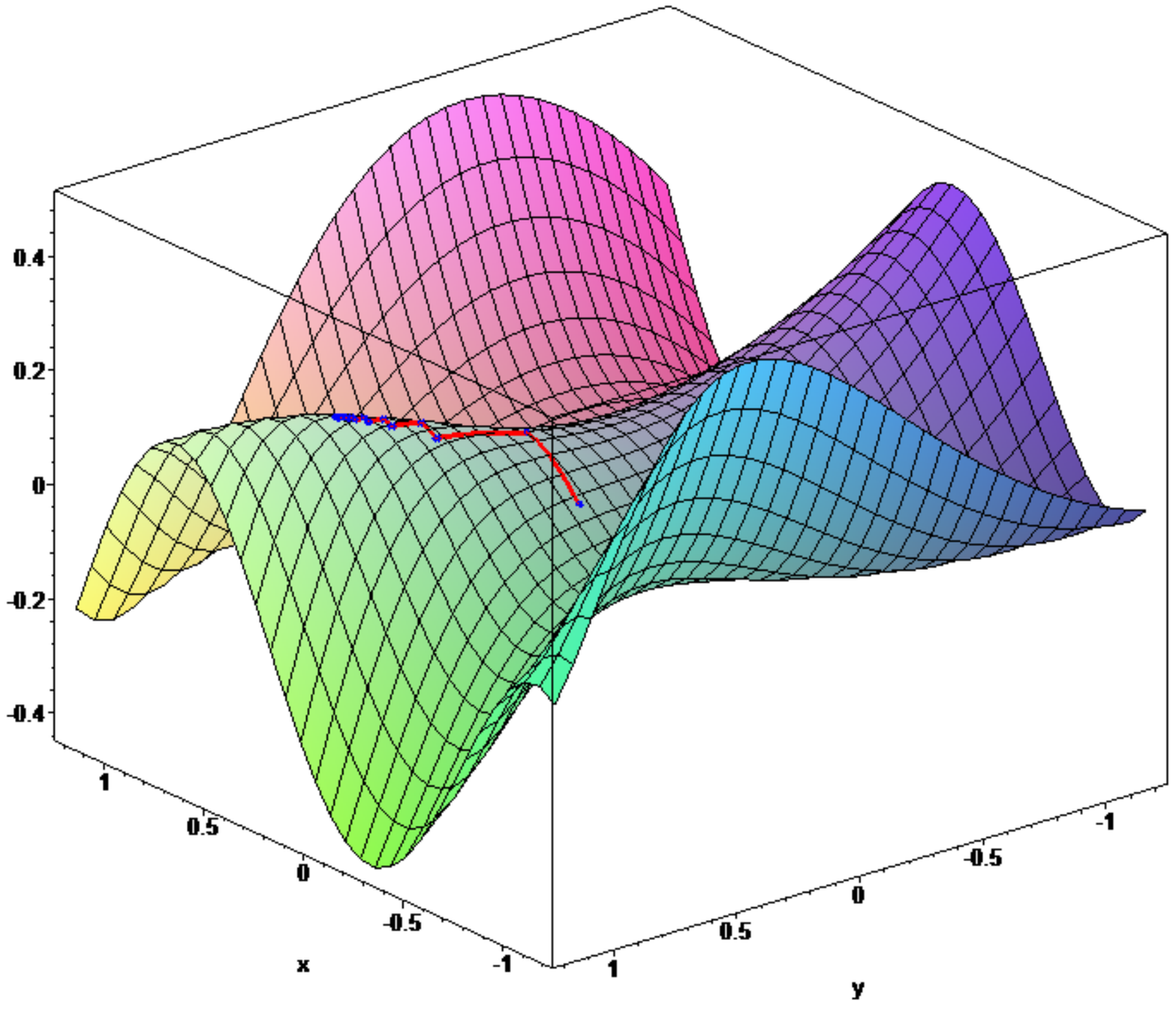

Cost of .numpy() call
Time (s)
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
0
200000
400000
600000
800000
1000000
Array size (elements)

```
x += 1
print(arr)

# array([[ 2.,  2.,  2.,  2.,  2.],
#        [ 2.,  2.,  2.,  2.,  2.],
#        [ 2.,  2.,  2.,  2.,  2.],
#        [ 2.,  2.,  2.,  2.,  2.],
#        [ 2.,  2.,  2.,  2.,  2.]], dtype=float32)

np.add(arr, 1, out=arr)
print(x)

# tensor([[3., 3., 3., 3., 3.],
#         [3., 3., 3., 3., 3.],
#         [3., 3., 3., 3., 3.],
#         [3., 3., 3., 3., 3.],
#         [3., 3., 3., 3., 3.]])
```

```
import PIL

arr = numpy.asarray(PIL.Image.open('dog.png'))
x = torch.from_numpy(arr)
print(x)

# tensor([[[-9.0190e-01,  2.9489e-01, -1.6344e+00,  ..., -1.3606e+00,
#            1.9089e+00, -4.2140e-02],
#          [ 5.2463e-01,  3.7772e-01,  1.9573e-01,  ..., -5.9366e-03,
#           -2.2412e-01,  1.6913e+00],
#          [-2.4837e-01, -1.3280e+00,  7.8298e-01,  ..., -5.8159e-02,
#            6.6103e-01, -2.0254e+00],
#          ...,
#          [-1.3256e+00, -4.6745e-01, -3.4172e-01,  ...,  1.4519e+00,
#            9.2960e-02, -3.8875e-01],
#          [ 2.6890e-01, -1.6123e+00,  1.1582e-01,  ..., -1.1667e+00,
#           -5.3774e-01, -3.8320e-01],
#          [-5.0798e-03,  2.1482e-01,  5.0101e-01,  ..., -1.2235e+00,
#            1.9274e+00,  3.3796e-01]],
#          ...])
```

High-performance Automatic Differentiation




0.4
0.2
0
-0.2
-0.4
1
0.5
0
-0.5
-1
x
1
0.5
0
-0.5
-1
y

```python
import torch

#  f(x) = x^2 + 5x + 4
# f'(x) =       2x + 5
def f(x):
    return x ** 2 + x * 5 + 4

x = torch.ones(1, requires_grad=True)
print(torch.autograd.grad(f(x), x))

# tensor([7.]) == f'(1)
```

```
import torch

def model(x, W, b):
    return torch.matmul(x, W) + b

data, targets = ...
W = torch.randn(5, 1, requires_grad=True)
b = torch.randn(1, requires_grad=True)

for example, target in zip(data, targets):
    output = model(example, W, b)
    loss = (output - target) ** 2

    grad_W, grad_b = torch.autograd.grad(loss, (W, b))

    W.data -= grad_W * lr
    b.data -= grad_b * lr
```

```
import torch

def model(x, W, b):
    return torch.matmul(x, W) + b

data, targets = ...
W = torch.randn(5, 1, requires_grad=True)
b = torch.randn(1, requires_grad=True)

for example, target in zip(data, targets):
    output = model(example, W, b)
    loss = (output - target) ** 2

    grad_W, grad_b = torch.autograd.grad(loss, (W, b))

    W.data -= grad_W * lr
    b.data -= grad_b * lr
```

```
import torch

def model(x, W, b):
    return torch.matmul(x, W) + b

data, targets = ...
W = torch.randn(5, 1, requires_grad=True)
b = torch.randn(1, requires_grad=True)

for example, target in zip(data, targets):
    W.grad.data.zero_()
    b.grad.data.zero_()

    output = model(example, W, b)
    loss = (output - target) ** 2

    loss.backward()

    W.data -= W.grad.data * lr
    b.data -= b.grad.data * lr
```

```
import torch

def model(x, W, b):
    return torch.matmul(x, W) + b

data, targets = ...
W = torch.randn(5, 1, requires_grad=True)
b = torch.randn(1, requires_grad=True)
optimizer = torch.optim.Adam([W, b])

for example, target in zip(data, targets):
    optimizer.zero_grad()

    output = model(example, W, b)
    loss = (output - target) ** 2

    loss.backward()

    optimizer.step()
```

```
import torch

def model(x, W, b):
    return torch.matmul(x, W) + b

data, targets = ...
W = torch.randn(5, 1, requires_grad=True)
b = torch.randn(1, requires_grad=True)
optimizer = torch.optim.Adam([W, b])

for example, target in zip(data, targets):
    optimizer.zero_grad()

    output = model(example, W, b)
    loss = (output - target) ** 2

    loss.backward()

    optimizer.step()
```

```
import torch
import random


rows, cols = 100, 100
x = torch.zeros(rows, cols, requires_grad=True)


while x.data.sum() < 10:                    # Data-dependent control flow!
    idx = random.randrange(rows)
    idx_var = torch.tensor([idx])
    x[idx] = row_model(idx_var)             # In-place modifications!
    x[idx].div_(x[idx].norm())


output = model(x)
loss = loss_fn(output, target)
loss.backward()
```

Higher-level helpers

✅ ML layers/ops

$$f(x, \theta)$$

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)
```

```
model.parameters()                                  # Iterator over parameters

model.state_dict()                                  # Weight serialization
model.load_state_dict(serialized_weights)           # and loading

model.eval()                                        # Train/eval mode toggle
model.train()

model.half()                                        # Type casts

model.zero_grad()                                   # Gradient management

model.cuda()                                        # Device changes

model = nn.DataParallel(model)                      # Multi-GPU

model = DistributedDataParallel(model)              # Multi-GPU + Multi-Node
```

✅ Data loading

- Vision
  - MNIST
  - Fashion MNIST
  - COCO (captioning and detection)
  - LSUN Classification
  - ImageFolder (generic classification dataset format)
  - Imagenet-12
  - CIFAR10 and CIFAR100
  - STL10
  - SVHN
  - PhotoTour
- Text
  - SST (sentiment analysis)
  - IMDb (sentiment analysis)
  - TREC (question classification)
  - SNLI (entailment)
  - Wikitext-2 (language modeling)
  - Abstract/generic support for machine translation

```
if opt.dataset in ['imagenet', 'folder', 'lfw']:
    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)),
                           ]))
elif opt.dataset == 'fake':
    dataset = dset.FakeData(image_size=(3, opt.imageSize, opt.imageSize),
                            transform=transforms.ToTensor())
```

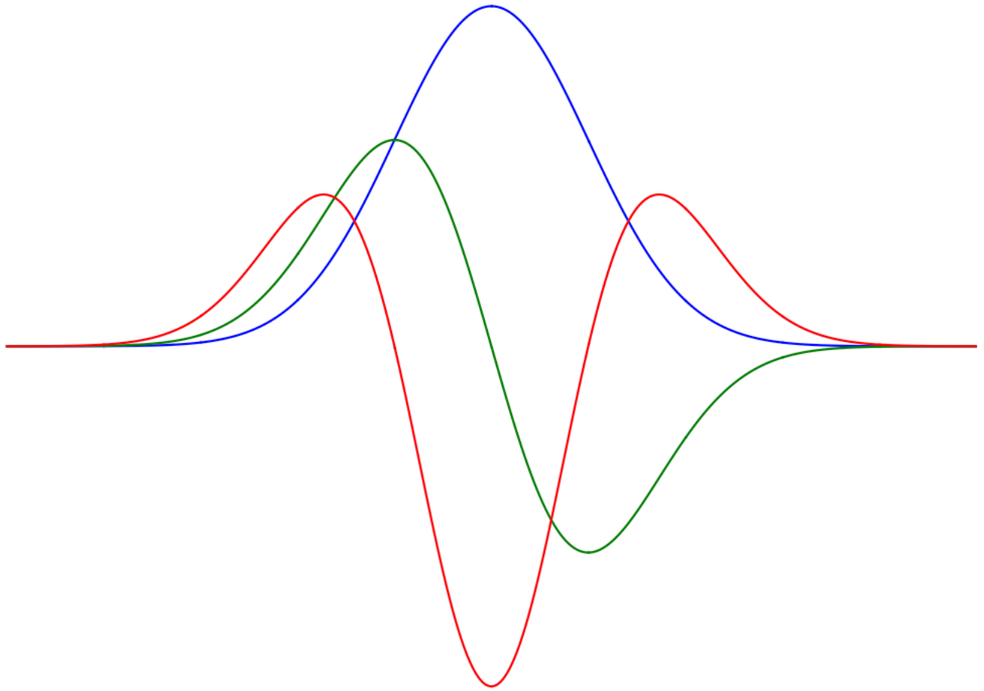
```
class MNIST(torch.utils.data.Dataset):
    def __init__(self, root, train=True, transform=None, target_transform=None):
        self.root = os.path.expanduser(root)
        self.transform = transform
        self.target_transform = target_transform

        self.download()
        self.data, self.labels = torch.load(
            os.path.join(self.root, self.processed_folder, MNIST.training_file))

    def __getitem__(self, index):
        img, target = self.data[index], self.labels[index]

        # doing this so that it is consistent with all other datasets to return a PIL Image
        img = Image.fromarray(img.numpy(), mode='L')

        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.data)
```

```
train_loader = torch.utils.data.DataLoader(
        train_dataset, batch_size=args.batch_size, shuffle=True)
        num_workers=args.workers, pin_memory=True)

for data_batch, label_batch in train_loader:
    ...  # train!
```

A single class for:
- multiprocess parallel data loading
- shuffling
- batching
- memory locking (for faster CUDA transfers)

✅ (More of) Python integration

```
class NormPdf(Function):
    @staticmethod
    def forward(ctx, x, loc=0.0, scale=1.0):
        y = torch.from_numpy(scipy.stats.norm.pdf(x))
        ctx.loc, ctx.scale = loc, scale
        ctx.save_for_backward(x, y)
        return y

    @staticmethod
    def backward(ctx, grad_y):
        x, y = ctx.saved_variables
        norm_x = (x - ctx.loc) / ctx.scale ** 2
        return -grad_y * y * norm_x

norm_pdf = NormPdf.apply

######################################################

x = torch.linspace(-5, 5, 1000,
                   requires_grad=True)


y   = norm_pdf(x)
dy  = torch.autograd.grad(y.sum(), x)
ddy = torch.autograd.grad(grad_y.sum(), x)
```



✅ Deployment

```
import torch
from torchvision import models

model = models.resnet18()
example_input = torch.randn(1, 3, 224, 224)
torch.onnx.export(model, example_input, 'resnet18.onnx')
```

Use the onnx pip/conda package to load and run on different backends (Caffe2, CNTK, hardware).

```python
import onnx

model = onnx.load("output/alexnet.onnx") # Load the ONNX model
onnx.checker.check_model(model)          # Check that the IR is well formed
print(onnx.helper.printable_graph(model.graph))
```

```
graph torch-jit-export (
  %0[FLOAT, 1x3x224x224]
) initializers (
  %1[FLOAT, 64x3x11x11]
  %2[FLOAT, 64]
  ...
  %15[FLOAT, 1000x4096]
  %16[FLOAT, 1000]
) {
  %17 = Conv[dilations = [1, 1], group = 1, kernel_shape = [11, 11], pads = [2, 2, 2, 2], strides = [4, 4]](%0, %1)
  %18 = Add[axis = 1, broadcast = 1](%17, %2)
  %19 = Relu(%18)
  %20 = MaxPool[kernel_shape = [3, 3], pads = [0, 0], strides = [2, 2]](%19)
  %21 = Conv[dilations = [1, 1], group = 1, kernel_shape = [5, 5], pads = [2, 2, 2, 2], strides = [1, 1]](%20, %3)
  %22 = Add[axis = 1, broadcast = 1](%21, %4)
  %23 = Relu(%22)
  ...
  %34 = MaxPool[kernel_shape = [3, 3], pads = [0, 0], strides = [2, 2]](%33)
  %35 = Reshape[shape = [1, 9216]](%34)
  %36, %37 = Dropout[is_test = 1, ratio = 0.5](%35)
  %38 = Transpose[perm = [1, 0]](%11)
  %40 = Gemm[alpha = 1, beta = 1, broadcast = 1](%36, %38, %12)
  %41 = Relu(%40)
  ...
  %50 = Gemm[alpha = 1, beta = 1, broadcast = 1](%47, %48, %16)
  return %50
}
```

Requires `pip install onnx-caffe2`.

```
import onnx
import onnx_caffe2.backend

img = np.random.randn(1, 3, 224, 224).astype(np.float32)

model = onnx.load('assets/squeezenet.onnx')

outputs = onnx_caffe2.backend.run_model(model, [img])
```

Easy C++ extensions

```
import torch

def jacobi_solve(A, b, iters):

    iter_matrix = -A.clone()
    iter_matrix.as_strided((A.size(0),), (A.size(0) + 1,)).zero_()
    iter_matrix /= A.diag().unsqueeze(-1)

    iter_offset = b / A.diag()

    cand = torch.zeros_like(b)
    for i in range(iters):
        cand = torch.matmul(iter_matrix, candidate) + iter_offset
    return cand




A, b = ...
print(jacobi_solve(A, b, 10000))
```

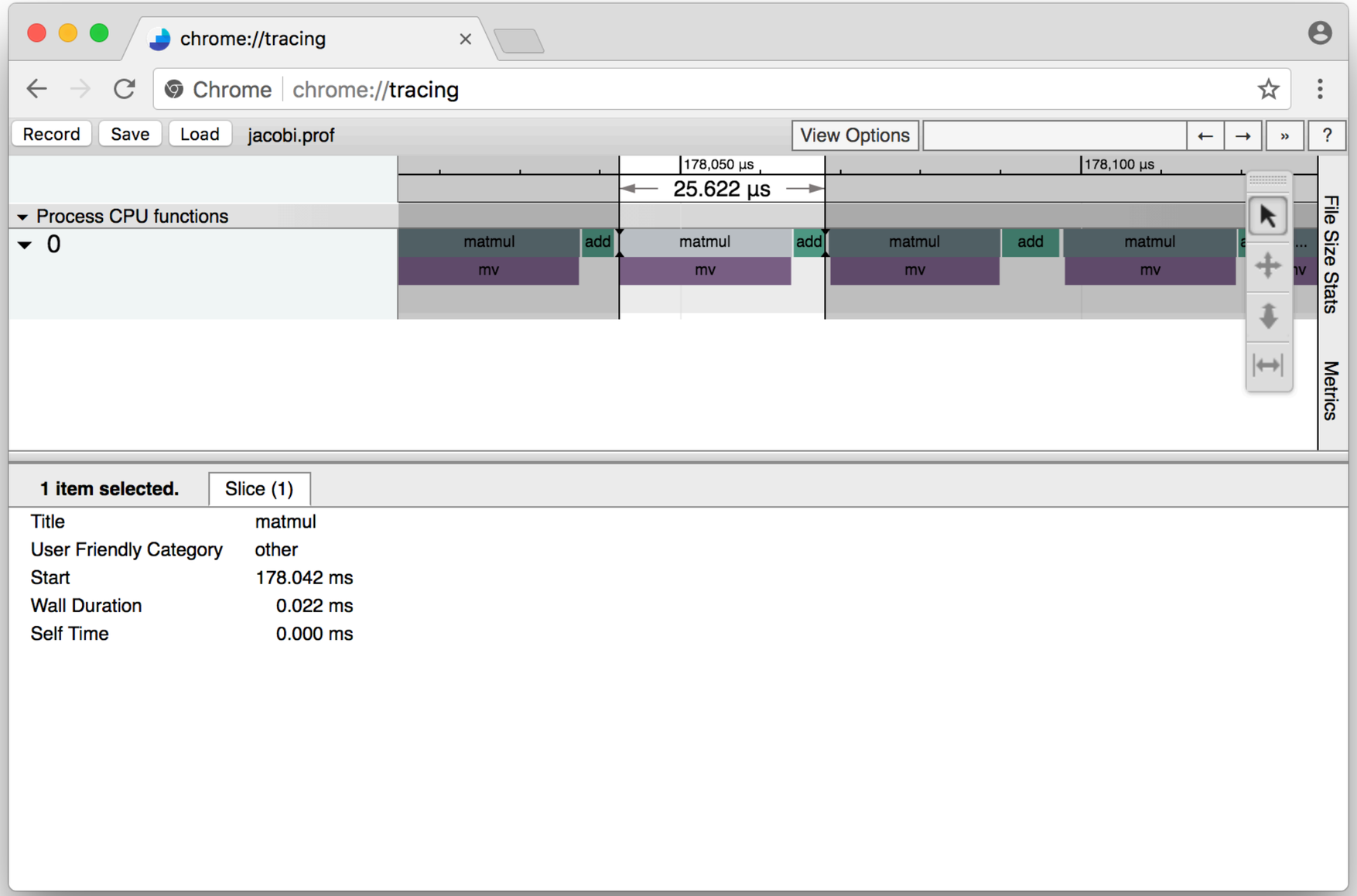
```cpp
#include <torch/torch.h>

torch::Tensor jacobi_solve(
        torch::Tensor A, torch::Tensor b, size_t iters) {
    auto iter_matrix = -A.clone();
    iter_matrix.as_strided({A.size(0)}, {A.size(0) + 1}).zero_();
    iter_matrix /= A.diag().unsqueeze(-1);

    auto iter_offset = b / A.diag();

    auto cand = torch::zeros_like(b);
    for (size_t i = 0; i < iters; ++i)
        cand = torch::matmul(iter_matrix, candidate) + iter_offset
    return cand;
}

PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
    m.def("jacobi_solve", &jacobi_solve,
          "Solves a square system of linear equations");
}
```

```
import torch
import torch.utils.cpp_extension


ext = torch.utils.cpp_extension.load(
    name='my_extension',
    sources=['extension.cpp'],
    extra_cflags=['-O3'])
jacobi_solve = ext.jacobi_solve


A, b = ...
print(jacobi_solve(A, b, 10000))
```

```
import torch
import torch.utils.cpp_extension


ext = torch.utils.cpp_extension.load(
    name='my_extension',
    sources=['extension.cpp'],
    extra_cflags=['-O3'])
jacobi_solve = ext.jacobi_solve


A, b = ...
torch.autograd.grad(
    jacobi_solve(A, b, 10000).sum(), b)
```

Profiler

```
import torch
from torch.autograd.profiler import profile

ext = ...
jacobi_solve = ...
A, b = ...


with profile() as p:
    jacobi_solve(A, b, 10000)

p.export_chrome_trace('jacobi.prof')
```


chrome://tracing
Chrome
chrome://tracing
Record
Save
Load
jacobi.prof
View Options
178,050 µs
178,100 µs
25.622 µs
Process CPU functions
0
matmul
add
mv
File Size Stats
Metrics
1 item selected.
Slice (1)
Title
matmul
User Friendly Category
other
Start
178.042 ms
Wall Duration
0.022 ms
Self Time
0.000 ms

Distributed computing

Simple interface, based on MPI:

```
import torch.distributed as dist

dist.init_process_group(backend='gloo',
                        init_method='file:///mount/shared/pt/syncfile',
                        world_size=4)
```

Simple interface, based on MPI:

```
import torch.distributed as dist

dist.init_process_group(backend='gloo',
                        init_method='file:///mount/shared/pt/syncfile',
                        world_size=4)
```

Backends:

- gloo
- MPI
- (soon) NCCL 2

Simple interface, based on MPI:

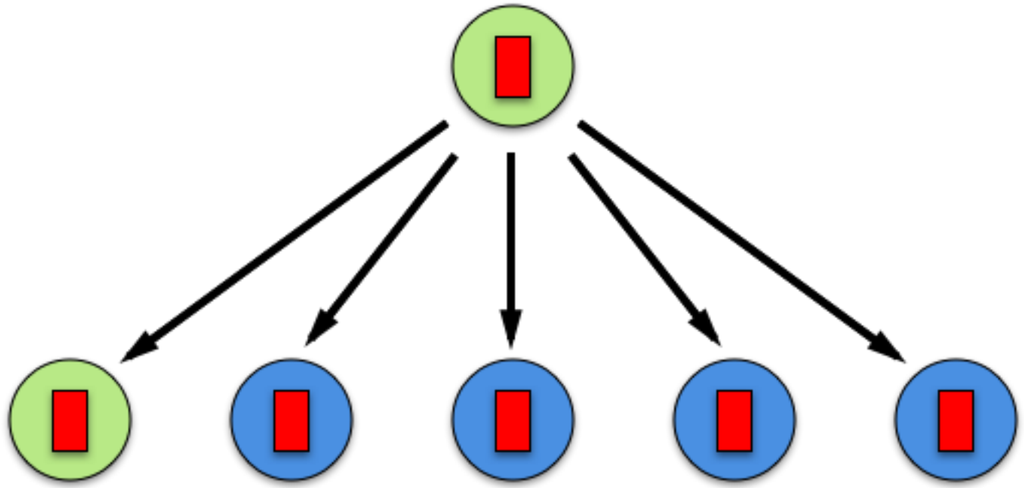
```
import torch.distributed as dist

dist.init_process_group(backend='gloo',
                        init_method='file:///mount/shared/pt/syncfile',
                        world_size=4)
```

Init methods:

- shared file system
- IP multicast
- chosen node
- environment

Collectives

- reduce
- all reduce
- broadcast
- scatter
- gather
- all gather
- barrier
- (i)send, (i)recv



DataParallel helper

```
import torch
import torch.nn as nn
import torch.distributed as dist
```

```
model = create_model()

data_loader = create_data_loader()

for batch, label in data_loader:
    y = model(batch)
    loss = loss_fn(y, label)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

DataParallel helper

```
import torch
import torch.nn as nn
import torch.distributed as dist
from torch.utils.data.distributed import DistributedSampler

dist.init_process_group(...)

model = create_model()
model = nn.parallel.DistributedDataParallel(model)
data_loader = create_data_loader(sampler=DistributedSampler(dset))

for batch, label in data_loader:
    y = model(batch)
    loss = loss_fn(y, label)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Released Jan 2017

200,000+ downloads

4500+ community repos

26,700+ user posts on forums

420 contributors

With ❤ from


facebook


NVIDIA


salesforce


ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIE
PARIS INSTITUTE OF TECHNOLOGY


Carnegie
Mellon
University


UNIVERSITE
PIERRE & MARIE CURIE
LA SCIENCE A PARIS


UBER


Stanford
University


UNIVERSITY OF
OXFORD


NYU


Inria


1794
ENS
ÉCOLE NORMALE
SUPÉRIEURE


VisionLabs
MACHINES CAN SEE


Digital
Reasoning