FROM RESEARCH TO PRODUCTION WITH PYTORCH 1.0

PETER GOLDSBOROUGH
I've been using PyTorch a few months now and I've never felt better. I have more energy. My skin is clearer. My eye sight has improved.
WHAT IS PYTORCH?

DYNAMIC NEURAL NETWORKS WITH STRONG GPU ACCELERATION
A MACHINE LEARNING FRAMEWORK BORN WITH AN EMPHASIS ON

- Dynamic Neural Networks
- Distributed Training
- GPU Acceleration
- Simplicity Over Complexity
- Eager Execution
PyTorch enables . . .
MISSION

PyTorch enables . . .

Research

252 Mentions @ ICLR 2018
(87 in 2018)
MISSION

PyTorch enables . . .

Research

Ecosystems

Allen NLP
PyTorch enables ...
PyTorch enables...

Research
Ecosystems
Partnerships
A MACHINE LEARNING FRAMEWORK
BORN WITH AN EMPHASIS ON

- EAGER EXECUTION
- DYNAMIC NEURAL NETWORKS
- GPU ACCELERATION
- DISTRIBUTED TRAINING
- SIMPLICITY OVER COMPLEXITY
A MACHINE LEARNING FRAMEWORK BORN WITH AN EMPHASIS ON

- EAGER EXECUTION
- DYNAMIC NEURAL NETWORKS
- DISTRIBUTED TRAINING
- GPU ACCELERATION
- SIMPLICITY OVER COMPLEXITY
Matrix a = ...;
Matrix b = ...;
Matrix c = ...;
scalar s = 7;

d = s * a + b;
e = matmul(c, d);

if s > 0 {
    x = d;
} else {
    x = e;
}

while s > 0 {
    x = input();
    c = matmul(c, x);
}

result = c;

Matrix<6, 9> a = ...;
Matrix<6, 9> b = ...;
Matrix<16, 6> c = ...;
scalar s = 7;

d = s * a + b;
e = matmul(c, d);

x = if_clause(s > 0, d, e);

while_loop(s > 0, [x, c], lambda x,c {
    x = input();
    c = matmul(c, x);
});

result = evaluate(x);
**PRO**
- No boundaries on flexibility
- Happier debugging
- No expensive compilation

**CONTRA**
- Harder to optimize
- Harder to deploy
<table>
<thead>
<tr>
<th><strong>PRO</strong></th>
<th><strong>CONTRA</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>No boundaries on flexibility</td>
<td>Harder to optimize</td>
</tr>
<tr>
<td>Happier debugging</td>
<td>Harder to understand</td>
</tr>
<tr>
<td>No expensive compilation</td>
<td>Harder to debug</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>EAGER</strong></th>
<th><strong>STATIC</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Easier to optimize</td>
<td>Easier to optimize</td>
</tr>
<tr>
<td>Easier to deploy</td>
<td>Easier to deploy</td>
</tr>
<tr>
<td>Easier to post-process</td>
<td>Easier to post-process</td>
</tr>
<tr>
<td>Harder to optimize</td>
<td>Harder to understand</td>
</tr>
<tr>
<td>Harder to deploy</td>
<td>Harder to debug</td>
</tr>
<tr>
<td></td>
<td>Less flexible</td>
</tr>
</tbody>
</table>
PYTORCH 1.0

A SEAMLESS PATH FROM RESEARCH TO PRODUCTION WITH TORCH SCRIPT
PyTorch

Models are Python programs

- Intuitive, Native
- Debuggable — `print` and `pdb`
- Hackable — use any Python library
PyTorch

Models are Python programs

- Simple
- Debuggable — `print` and `pdb`
- Hackable — use any Python library

- Needs Python to run
- Difficult to optimize and parallelize
PyTorch *Eager Mode*

Models are Python programs

- Simple
- Debuggable — `print` and `pdb`
- Hackable — use any Python library

- Needs Python to run
- Difficult to optimize and parallelize
PyTorch  *Eager Mode*

Models are Python programs

- Simple
- Debuggable — `print` and `pdb`
- Hackable — use any Python library

- Needs Python to run
- Difficult to optimize and parallelize

---

PyTorch  *Script Mode*

Models are programs written in an optimizable subset of Python

- Production deployment
- No Python dependency
- Optimizable
Tools to transition eager code into script mode

**EAGER MODE**
For prototyping, training, and experiments

**SCRIPT MODE**
For use at scale in production

- torch.jit.script
- torch.jit.trace
Transitioning a model 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 Torch Script module.
- Can reuse existing eager model code
- ⚠ Control-flow is ignored

```python
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))
```
Transitioning a model with `@torch.jit.script`

Write model directly in a subset of Python, annotated 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.

```python
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
```

You can mix both trace and script in a single model.
Loading a model without Python

Torch Script models can be saved to a model archive, and loaded in a python-free executable using a C++ API.

Our C++ Tensor API is the same as our Python API, so you can do preprocessing and post processing before calling the model.

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

// C++: load and run model
auto module = torch::jit::load("serialized_resnet.pt");
auto example = torch::rand({1, 3, 224, 224});
auto output = module->forward({example}).toTensor();
std::cout << output.slice(1, 0, 5) << 'n';
```
What subset of PyTorch is valid Torch Script?

✓ Tensors and numeric primitives
✓ If statements
✓ Simple loops
✓ Code organization using `nn.Module`
✓ Tuples, Lists
✓ `print` and strings
✓ Gradients propagation through script functions

✗ In-place updates to tensors or lists
✗ Direct use of standard `nn.Modules` like `nn.Conv` (trace them instead!)
✗ Calling `grad()` or `backwards()` within `@script`

For more details [https://pytorch.org/docs/master/jit.html#torch-script-language-reference](https://pytorch.org/docs/master/jit.html#torch-script-language-reference)
PYTORCH IN C++

FLEXIBILITY
SIMPLICITY
PERFORMANCE
ACROSS LANGUAGE BOUNDARIES
```cpp
#include <torch/csrc/autograd/variable.h>
#include <torch/csrc/autograd/function.h>

at::Tensor x = torch::ones({2, 3}, torch::requires_grad());
at::Tensor y = torch::randn({2, 3});

auto z = x + y;
z.backward();
std::cout << x.grad();
```
The power of C++, CUDA and ATen in imperative PyTorch models
#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);
}
from setuptools import setup
from torch.utils.cpp_extension import 
    import BuildExtension, CppExtension

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

import torch.utils.cpp_extension

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

module.compute(...)

SETUPTOOLS

JIT EXTENSION
import torch
import extension

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

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
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");
}

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")