How PyTorch Scales Deep Learning from Experimentation to Production

Vincent Quenneville-Bélair, PhD. Facebook AI.
Overview

Compute with PyTorch

Model with Neural Networks

Ingest Data

Use Multiple GPUs and Machines
Compute with PyTorch
def pairwise_distance(a, b):
    p = a.shape[0]
    q = b.shape[0]
    squares = torch.zeros((p, q))
    for i in range(p):
        for j in range(q):
            diff = a[i, :] - b[j, :]
            diff_squared = diff ** 2
            squares[i, j] = torch.sum(diff_squared)
    return squares

a = torch.randn(100, 2)
b = torch.randn(200, 2)

%timeit pairwise_distance(a, b)
# 438 ms ± 16.7 ms per loop
Example: Batched Pairwise Distance

```python
def pairwise_distance(a, b):
    diff = a[:, None, :] - b[None, :, :]  # Broadcast
    diff_squared = diff ** 2
    return torch.sum(diff_squared, dim=2)

a = torch.randn(100, 2)
b = torch.randn(200, 2)

%timeit pairwise_distance(a, b)
# 322 µs ± 5.64 µs per loop
```
%timeit, print, pdb

torch.utils.bottleneck

also pytorch.org/docs/stable/jit.html#debugging
Script for Performance

**Eager mode:** PyTorch – Models are simple debuggable python programs for prototyping

**Script mode:** TorchScript – Models are programs converted and ran by lean Just-In-Time interpreter in production
a = torch.rand(5)
def func(x):
    for i in range(10):
        x = x * x
    return x

scripted_func = torch.jit.script(func)

%timeit func(a)
# 18.5 µs ± 229 ns per loop

%timeit scripted_func(a)
# 4.41 µs ± 26.5 ns per loop
Just-In-Time Intermediate Representation

```
scripted_func.graph_for(a)

# graph(%x.1 : Float(*)):
#   %x.15 : Float(*) = prim::FusionGroup_0(%x.1)
#   return (%x.15)
# with prim::FusionGroup_0 = graph(%18 : Float(*)):
#   %x.4 : Float(*) = aten::mul(%18, %18) # <ipython-input-13-1ec87869e140>:3:12
#   %x.5 : Float(*) = aten::mul(%x.4, %x.4) # <ipython-input-13-1ec87869e140>:3:12
#   %x.6 : Float(*) = aten::mul(%x.5, %x.5) # <ipython-input-13-1ec87869e140>:3:12
#   %x.9 : Float(*) = aten::mul(%x.6, %x.6) # <ipython-input-13-1ec87869e140>:3:12
#   %x.10 : Float(*) = aten::mul(%x.9, %x.9) # <ipython-input-13-1ec87869e140>:3:12
#   %x.11 : Float(*) = aten::mul(%x.10, %x.10) # <ipython-input-13-1ec87869e140>:3:12
#   %x.12 : Float(*) = aten::mul(%x.11, %x.11) # <ipython-input-13-1ec87869e140>:3:12
#   %x.13 : Float(*) = aten::mul(%x.12, %x.12) # <ipython-input-13-1ec87869e140>:3:12
#   %x.14 : Float(*) = aten::mul(%x.13, %x.13) # <ipython-input-13-1ec87869e140>:3:12
#   %x.15 : Float(*) = aten::mul(%x.14, %x.14) # <ipython-input-13-1ec87869e140>:3:12
#   return (%x.15)

scripted_func.save("func.pt")
```
Performance Improvements

**Algebraic rewriting** – Constant folding, common subexpression elimination, dead code elimination, loop unrolling, 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** – Compiling parts of the program for specific hardware. Integration also ongoing with TVM, Halide, Glow, XLA

**Runtime** – No python global interpreter lock. Fork and wait parallelism.
Model with Neural Networks
class Net(torch.nn.Module):

    def __init__(self):
        ...

    def forward(self, x):
        ...

model = Net()
print(model)

# Net(
#   (conv1): Conv2d(1, 6, kernel_size=(3, 3), stride=(1, 1))
#   (conv2): Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1))
#   (fc1): Linear(in_features=576, out_features=120, bias=True)
#   (fc2): Linear(in_features=120, out_features=84, bias=True)
#   (fc3): Linear(in_features=84, out_features=10, bias=True)
# )
class Net(torch.nn.Module):

    def __init__(self):
        ...

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))
        x = x.view(-1, 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(x):
        return math.prod(x.size()[1:])
Optimize with SGD. Differentiate with Autograd.
Training Loop

```python
from torch.optim import SGD

loader = ...
model = Net()
criterion = torch.nn.CrossEntropyLoss()  # LogSoftmax + NLLLoss
optimizer = SGD(model.parameters)

for epoch in range(10):
    for batch, labels in loader:
        outputs = model(batch)
        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```
Ingest Data
Datasets

```python
class IterableStyleDataset(torch.utils.data.IterableDataset):
    
def __iter__(self):
        # Support for streams
        ...

class MapStyleDataset(torch.utils.data.Dataset):
    
def __getitem__(self, key):
        # Map from (non-int) keys
        ...

    def __len__(self):
        # Support sampling
        ...

# Preprocessing
```
from torch.utils.data import DataLoader, RandomSampler

dataloader = DataLoader(
    dataset,  # only for map-style
    batch_size=8,  # balance speed and convergence
    num_workers=2,  # non-blocking when > 0
    sampler=RandomSampler,  # random read may saturate drive
    pin_memory=True,  # page-lock memory for data?
)
Copy from host to GPU is faster from RAM directly. To prevent paging, pin tensor to page-locked RAM.

Once a tensor is pinned, use asynchronous GPU copies with `to(device, non_blocking=True)` to overlap data transfers with computation.

A single Python process can saturate multiple GPUs, even with the global interpreter lock.
Copy from host to GPU is faster from RAM directly. To prevent paging, pin tensor to page-locked RAM.

Once a tensor is pinned, use asynchronous GPU copies with `to(device, non_blocking=True)` to overlap data transfers with computation.

A single Python process can saturate multiple GPUs, even with the global interpreter lock.
Use Multiple GPUs and Machines
Data Parallel – Data distributed across devices

Model Parallel – Model distributed across devices
Single Machine Data Parallel
Single Machine Model Parallel
Distributed Data Parallel
Distributed Data Parallel with Model Parallel
Distributed Model Parallel
Single Machine Data Parallel

scatter

replicate

gather

GPU0

GPU1

GPU2

GPU3

scatter

[10, 11, 12, 13]

10

m

m

m

m

o1

o2

o3

(loss)

[00, 01, 02, 03]

replicate

[10, 11, 12, 13]

11

m

m

m

m

o1

o2

o3

[00, 01, 02, 03]

gather

[10, 11, 12, 13]

12

m

m

m

m

o2

o3

o3

[00, 01, 02, 03]

loss

[00, 01, 02, 03]
model = Net().to("cuda:0")
model = torch.nn.DataParallel(model)

# training loop ...
Single Machine Model Parallel
class Net(torch.nn.Module):
    def __init__(self, *gpus):
        super(Net).__init__(self)
        self.gpu0 = torch.device(gpus[0])
        self.gpu1 = torch.device(gpus[1])
        self.sub_net1 = torch.nn.Linear(10, 10).to(self.gpu0)
        self.sub_net2 = torch.nn.Linear(10, 5).to(self.gpu1)
    def forward(self, x):
        y = self.sub_net1(x.to(self.gpu0))
        z = self.sub_net2(y.to(self.gpu1))  # blocking
        return z

model = Net("cuda:0", "cuda:1")

# training loop...
```python
def one_machine(machine_rank, world_size, backend):
    torch.distributed.init_process_group(
        backend, rank=machine_rank, world_size=world_size
    )

gpus = {
    0: [0, 1],
    1: [2, 3],
}[machine_rank]  # or one gpu per process to avoid GIL

model = Net().to(gpus[0])  # default to first gpu on machine
model = torch.nn.parallel.DDP(model, device_ids=gpus)

# training loop...

for machine_rank in range(world_size):
    torch.multiprocessing.spawn(
        one_machine, args=(world_size, backend),
        nprocs=world_size, join=True  # blocking
    )
```
Distributed Data Parallel with Model Parallel
Distributed Data Parallel with Model Parallel

```python
def one_machine(machine_rank, world_size, backend):
    torch.distributed.init_process_group(
        backend, rank=machine_rank, world_size=world_size
    )

gpus = {
    0: [0, 1],
    1: [2, 3],
}[machine_rank]

model = Net(gpus)
model = torch.nn.parallel.DDP(model)

# training loop...

for machine_rank in range(world_size):
    torch.multiprocessing.spawn(
        one_machine, args=(world_size, backend),
        nprocs=world_size, join=True
    )
```
Distributed Model Parallel (in development)
Distributed Model Parallel (in development)
Conclusion
Scale from experimentation to production.
Questions?
Quantization (in development)

Replace `float32` by `int8` to save bandwidth

pytorch.org/docs/stable/quantization.html