

End-to-End Machine Learning Systems

Vincent Quenneville-Belair, Facebook AI

Overview

Compute with PyTorch

Model with Neural Networks

Fit the Best Model

Load Data

Use Multiple GPUs and Machines

Machine Learning Pipeline Plumbing

Data ingestion

Data preparation

Model experimentation

Model training

Model evaluation

Model deployment

Model scaling

Performance monitoring

Compute with PyTorch

Batching: pairwise distance

```
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
```

Batching: pairwise distance

```
def pairwise_distance(a, b):
    diff = a[:, None, :] - b[None, :, :]
    # Broadcasting
    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
```

Script for Performance

Eager mode: PyTorch

Models are simple debuggable python programs for prototyping

Script mode: TorchScript

Models are programs ran by lean JIT interpreter in production

From Eager to Script Mode

```
a = torch.rand(5)
def func(x):
    for i in range(10):
        x = x * x
    return x

# often used as decorator
scripted_func = torch.jit.script(func) # see also trace

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

%timeit scripted_func(a)
# 4.41 µs ± 26.5 ns per loop
```

JIT Intermediate Representation with Fused Operations

```
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-13-1ec878
#     %x.5 : Float(*) = aten::mul(%x.4, %x.4) # <ipython-input-13-1ec8
#     %x.6 : Float(*) = aten::mul(%x.5, %x.5) # <ipython-input-13-1ec8
#     %x.9 : Float(*) = aten::mul(%x.6, %x.6) # <ipython-input-13-1ec8
#     %x.10 : Float(*) = aten::mul(%x.9, %x.9) # <ipython-input-13-1ec
#     %x.11 : Float(*) = aten::mul(%x.10, %x.10) # <ipython-input-13-1
#     %x.12 : Float(*) = aten::mul(%x.11, %x.11) # <ipython-input-13-1
#     %x.13 : Float(*) = aten::mul(%x.12, %x.12) # <ipython-input-13-1
#     %x.14 : Float(*) = aten::mul(%x.13, %x.13) # <ipython-input-13-1
#     %x.15 : Float(*) = aten::mul(%x.14, %x.14) # <ipython-input-13-1
#     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

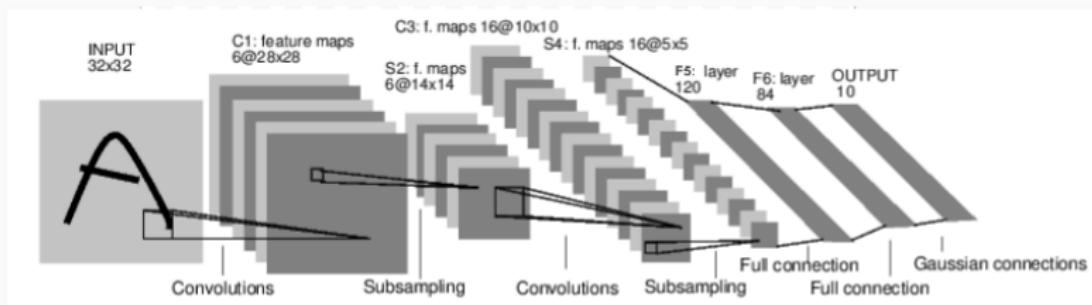
Compiling parts of the program for specific hardware. Integration ongoing with codegen frameworks: TVM, Halide, Glow, XLA

Runtime

No python global interpreter lock. Fork and wait parallelism.

Model with Neural Networks

Application to Vision



Neural Network

```
class Net(torch.nn.Module):

    def __init__(self):
        ...

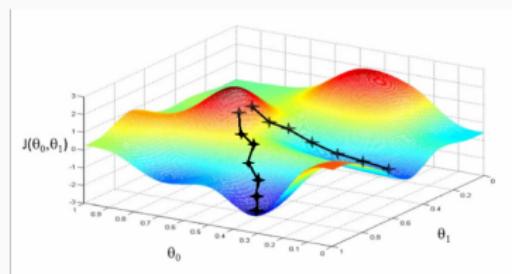
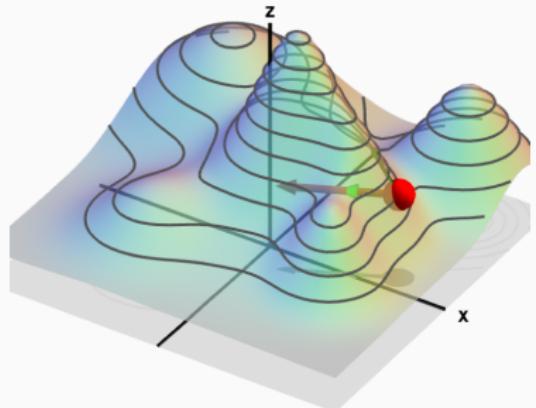
    def forward(self, x):
        ...

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)
# )
```

Fit the Best Model

Gradient Descent, $-df/dw$



GD to SGD

Minimize

$$L(w) = \frac{1}{n} \sum_i L_i(w)$$

Gradient Descent

$$w \leftarrow w - \alpha \frac{1}{n} \sum_i \frac{d}{dw} L_i(w)$$

Stochastic Gradient Descent

$$w \leftarrow w - \alpha \frac{d}{dw} L_i(w)$$

Test of time award!

GD to SGD

Minimize

$$L(w) = \frac{1}{n} \sum_i L_i(w)$$

Gradient Descent

$$w \leftarrow w - \alpha \frac{1}{n} \sum_i \frac{d}{dw} L_i(w)$$

Stochastic Gradient Descent

$$w \leftarrow w - \alpha \frac{d}{dw} L_i(w)$$

Test of time award!

How do we compute derivatives?

Backpropagation

The derivative of

$$y = f_3(f_2(f_1(x)))$$

is

$$\frac{dy}{dx} = \frac{df_3}{df_2} \frac{df_2}{df_1} \frac{df_1}{dx}$$

by chain rule

Example

We can write

$$h_{i+1} = \tanh(W_h h_i^T + W_x x^T)$$

as

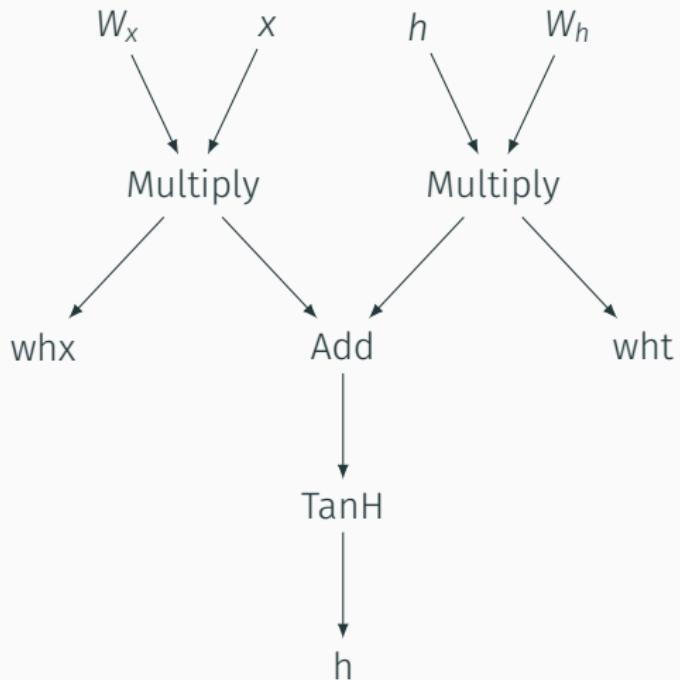
$$wht \leftarrow W_h h_i^T$$

$$whx \leftarrow W_x x^T$$

$$h \leftarrow wht + whx$$

$$h \leftarrow \tanh h$$

Example



Backward pass provides derivative

Training Loop

```
from torch.optim import SGD
from torch.optim.lr_scheduler import ExponentialLR

loader = ...
model = ...
optimizer = SGD(model.parameters())
scheduler = ExponentialLR(optimizer)

for epoch in range(10):
    for batch, label in loader:

        output = model(batch)
        loss = criterion(output, label)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    scheduler.step()
```

Load Data

Datasets

```
class IterableStyleDataset(torch.utils.data.IterableDataset):

    def __iter__(self):
        # Support for streams of unknown lengths
        ...

class MapStyleDataset(torch.utils.data.Dataset):

    def __getitem__(self):
        # Map from possibly non-int keys
        ...

    def __len__(self):
        # Support random sampling with torch.utils.data.Sampler
        ...
```

DataLoader

```
dataloader = torch.utils.data.DataLoader(  
    dataset,                      # only for map-style  
    batch_size=4,  
    sampler=torch.utils.data.RandomSampler  
    num_workers=4,                # non-blocking when > 0  
    pin_memory=True,              # save to page-locked memory  
    collate_fn=collate_fn,        # combine samples into batch  
)
```

Pinned Memory in DataLoader

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.

Pinned Memory in DataLoader

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

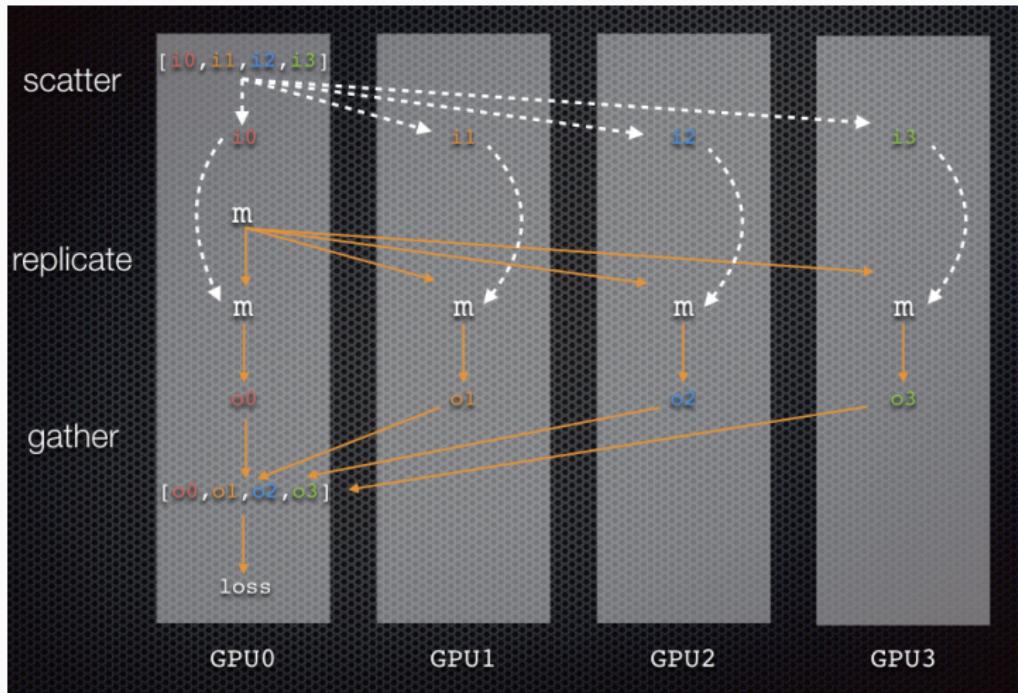
Single Machine Model Parallel

Distributed Data Parallel

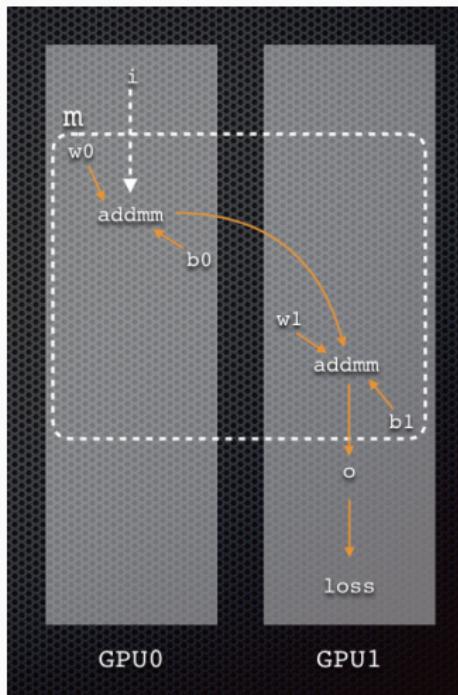
Distributed Data Parallel with Model Parallel

Distributed Model Parallel

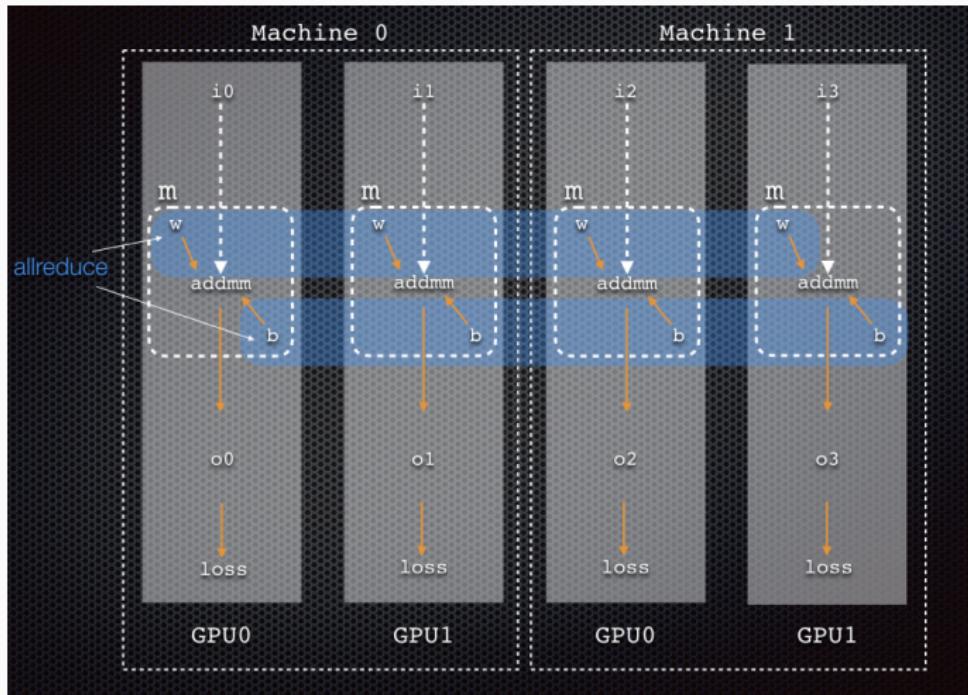
Data Parallel



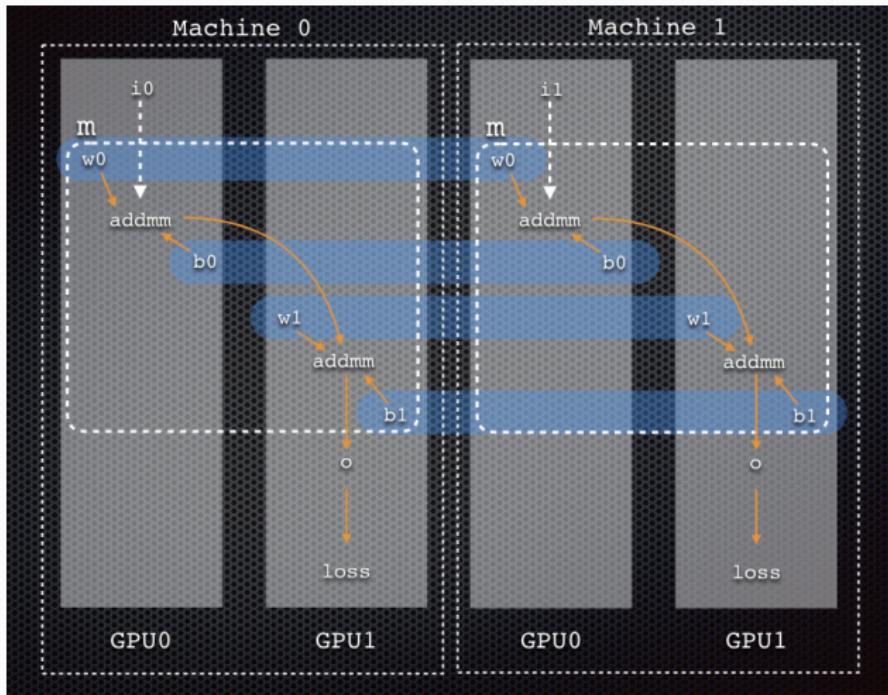
Distributed Data Parallel



Model Parallel



Distributed Data Parallel with Model Parallel



Distributed Model Parallel

in development

Data Parallel

```
model = Net().to("cuda:0")
model = torch.nn.DataParallel(model)
# preferred to torch.multiprocessing
```

Distributed Data Parallel

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

    device_ids = {
        0: [0, 1, 2, 3, 4],
        1: [5, 6, 7, 8, 9],
    }[rank]

    # device_ids = [rank] # One device per process to avoid GIL

    model = Net().to(device_ids[rank])
    model = torch.nn.parallel.DDP(model, device_ids=device_ids)

for rank in range(world_size):
    torch.multiprocessing.spawn(
        one_machine, args=(world_size, backend),
        nprocs=world_size, join=True
    )
```

Model Parallel

```
class Net(torch.nn.Module):

    def __init__(self, dev0, dev1):
        super(Net).__init__(self)
        self.dev0 = torch.device(dev0)
        self.dev1 = torch.device(dev1)

        self.sub_net1 = torch.nn.Linear(10, 10).to(self.dev0)
        self.sub_net2 = torch.nn.Linear(10,      5).to(self.dev1)

    def forward(self, x):
        y = self.sub_net1(x.to(self.dev0))
        z = self.sub_net2(y.to(self.dev1))
        return z

Net("cuda:0", "cuda:1")
```

Combine DDP with Model Parallel

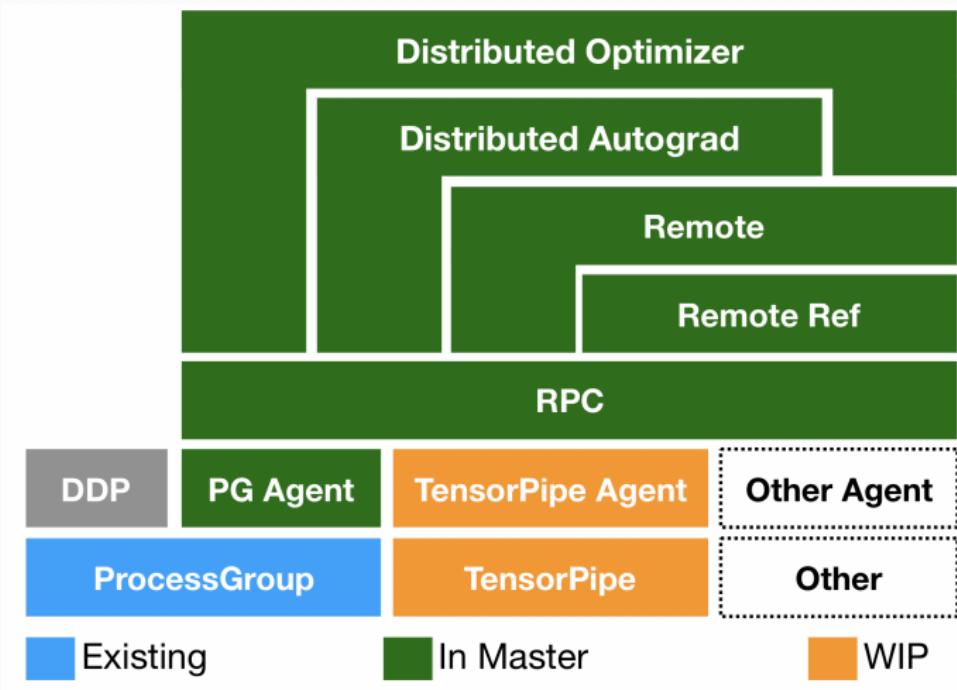
```
def one_machine(rank, world_size, backend):
    torch.distributed.init_process_group(
        backend, rank=rank, world_size=world_size
    )

    dev0, dev1 = {
        0: [0, 1], 1: [2, 3],
        3: [4, 5], 4: [6, 7], # on same machine or not
    }[rank]

    model = Net(dev0, dev1)
    model = torch.nn.parallel.DDP(model, device_ids=None)

for rank in range(world_size):
    torch.multiprocessing.spawn(
        one_machine, args=(world_size, backend),
        nprocs=world_size, join=True
    )
```

Model Parallel using RPC



in development

Summary

Summary

Compute with PyTorch

Model with Neural Networks

Fit the Best Model

Load Data

Use Multiple GPUs and Machines

Questions?