11-695: AI Engineering
Devices Management II

LTI/SCS

Spring 2020

Outline

Generic Models[1]

Carnegie Mellon

torch_general_model

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

class Mnist_CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(16, 16, kernel_size=3, stride=2, padding=1)
        self.conv3 = nn.Conv2d(16, 10, kernel_size=3, stride=2, padding=1)

    def forward(self, xb):
        xb = xb.view(-1, 1, 28, 28)
        xb = F.relu(self.conv1(xb))
        xb = F.relu(self.conv2(xb))
        xb = F.relu(self.conv3(xb))
        xb = F.avg_pool2d(xb, 4)
        return xb.view(-1, xb.size(1))
```

- Like `tf.keras`, we can either write the whole network as a class extending from `nn.Module`

[1] https://pytorch.org/tutorials/beginner/nn_tutorial.html

Generic Models

Carnegie Mellon

torch_general_model_2

```
class Lambda(nn.Module):
    def __init__(self, func):
        super().__init__()
        self.func = func
    def forward(self, x):
        return self.func(x)

model = nn.Sequential(
    Lambda(lambda x: x.view(-1, 1, 28, 28)),
    nn.Conv2d(1, 16, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.Conv2d(16, 16, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.Conv2d(16, 10, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.AvgPool2d(4),
    Lambda(lambda x: x.view(x.size(0), -1)),
)
```

- Or can use `nn.Sequential`

GradientTape + optimizer to autograd[2]

Carnegie Mellon

torch_autograd

```
1  model = torch.nn.Sequential(
2      torch.nn.Linear(D_in, H),
3      torch.nn.ReLU(),
4      torch.nn.Linear(H, D_out))
5  loss_fn = torch.nn.MSELoss(reduction='sum')
6
7  for t in range(epochs):
8      x, y = from_dataset(...)
9      y_pred = model(x)
10     loss = loss_fn(y_pred, y)
11
12     # backprop calculation
13     model.zero_grad() # manually zero out all Grads
14     loss.backward() # new Grads calculation
15
16     # SGD update
17     with torch.no_grad(): # meaning not tracking grads change
18         for param in model.parameters():
19             param -= learning_rate * param.grad
```

- They have a same purpose: auto-tracking grads

[2] https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

From autograd to optim

Carnegie Mellon

toch_optim

```
1  model = torch.nn.Sequential(
2      torch.nn.Linear(D_in, H),
3      torch.nn.ReLU(),
4      torch.nn.Linear(H, D_out))
5  loss_fn = torch.nn.MSELoss(reduction='sum')
6  opt = torch.optim.Adam(model.parameters(), lr=1e03)
7
8  for t in range(epochs):
9      x, y = from_dataset(...)
10     y_pred = model(x)
11     loss = loss_fn(y_pred, y)
12
13     # backprop calculation
14     optimizer.zero_grad() # Note: not model.zero_grad()
15     loss.backward() # same
16
17     # SGD update
18     opt.step() # simple
```

- Use `torch.optim` for convenience

Manual transfer between CPU ↔ GPU[3]

Carnegie Mellon

tf_devices

```
1  device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
2
3  # move model to device
4  net.to(device)
5
6  # move every input batch to device
7  inputs, labels = data[0].to(device), data[1].to(device)
8
9  # optimize as usual
10 loss  = criterion(model(inputs), labels)
11 ... # backgrop and SGD (more later)
12
13 # transfer back to CPU
14 loss.cpu().numpy() # Note: for weights w it usually has w.grad and w.data
15
16 # or along with recasting dtype
17 loss.to("cpu", torch.double))
```

- Remember to reside your model in GPUs first to accept inputs

[3] https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

Dynamic Graph[4]

Carnegie Mellon

torch_dynamic

```
class DynamicNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(DynamicNet, self).__init__()
        self.input_linear = torch.nn.Linear(D_in, H)
        self.middle_linear = torch.nn.Linear(H, H)
        self.output_linear = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """Reuse the middle_linear Module random times

        Here we also see that it is perfectly safe to reuse the same Module many
        times when defining a computational graph. This is a big improvement from Lua
        Torch, where each Module could be used only once."""
        h_relu = self.input_linear(x).clamp(min=0)
        for _ in range(random.randint(0, 3)):
            h_relu = self.middle_linear(h_relu).clamp(min=0)
        y_pred = self.output_linear(h_relu)
        return y_pred
```

- This dynamic design is one of the biggest advantages on Pytorch

[4] https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

From dataset to DataLoader[5]

Carnegie Mellon

torch_dataset

```
from torch.utils.data import DataLoader

# you have tensors already
train_ds = TensorDataset(x_train, y_train)
train_dl = DataLoader(train_ds, batch_size=32, shuffle=True, num_workers=4)

# or from a raw data folder
data_transform = transforms.Compose([
        transforms.RandomSizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])
my_dataset = datasets.ImageFolder(root='my_data/train',
                                                transform=data_transform)
dataset_loader = torch.utils.data.DataLoader(my_dataset,
                                               batch_size=4, shuffle=True,
                                               num_workers=4)
```

- Similar to `tf.keras`

[5] https://pytorch.org/tutorials/beginner/data_loading_tutorial.html

Train and Test

Carnegie Mellon

torch_train_eval

```
criterion = nn.CrossEntropyLoss()
opt = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
for epoch in range(epochs):
    model.train()
    for xb, yb in train_dl:
        pred = model(xb)
        loss = criterion(pred, yb)

        # almost always do this triplet:
        opt.zero_grad() # have to do this MANUALLY
        loss.backward() # backprop: calculate grads
        opt.step() # one SGD-based update step


    model.eval() # turn on this flag for evaluation/test
    with torch.no_grad(): # don't forget to turn off grads tracking as well
        valid_loss = sum(loss_func(model(xb), yb) for xb, yb in valid_dl)

    print(epoch, valid_loss / len(valid_dl))
```

- For training: `model.train()`
- Otherwise: `model.eval()`, with `torch.no_grad()`

Outline

Data Parallelism with Multiple GPUs[6]

Carnegie Mellon

torch_data_parallel_multiple_gpus

```
# feed data into a data loader as normal
train_ds = TensorDataset(x_train, y_train)
train_dl = DataLoader(train_ds, batch_size=32, shuffle=True, num_workers=4)

# define model as normal
model = MNIST_CNN() # see slide no. 3

# signify data parallelism
if torch.cuda.device_count() > 1:
    model = nn.DataParallel(model)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model.to(device)

for data in rand_loader:
    input = data.to(device) # 4 nodes will have 8 data samples each
    output = model(input)
```

- `pytorch` splits data equally for us
- It will collect and merge results before returning

[6] https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html

Outline

1. Py-torch for `tensorflow` users
2. Data Paralellism with `pytorch`
3. `torch.distributed`

Python Multiprocessing[7]

Carnegie Mellon

python_multiprocessing

```
1  from multiprocessing import Process
2
3  def f(pid):
4      print('hello ', pid)
5
6  if __name__ == '__main__':
7      ps = []
8      for i in range(5):
9          p = Process(target=f, args=(i,))
10         p.start()
11         ps.append(p)
12
13     for p in ps:
14         p.join() # wait until completion
```

- Usually each process will process a segment of a big job
- In advanced settings, processes can communicate with each other
- Restricted to one machine

[7] https://docs.python.org/3.6/library/multiprocessing.html

torch.distributed and .multiprocessing

Carnegie Mellon

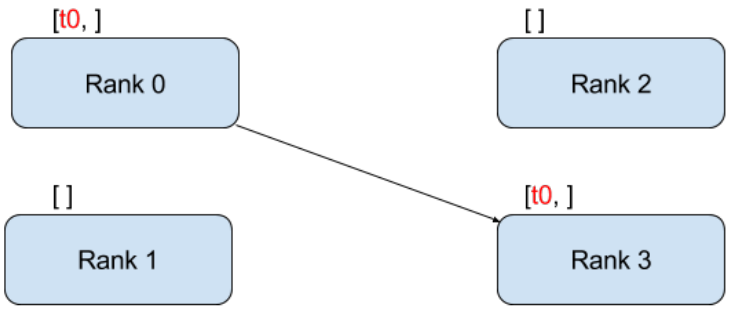
torch_dist_multiprocessing

```
1  import torch.distributed as dist
2  from torch.multiprocessing import Process
3
4  def run(rank, size): # to be passed into a single process later
5      pass
6
7  def init_process(rank, size, fn, backend='gloo'):
8      """ Initialize the distributed environment. """
9      os.environ['MASTER_ADDR'] = '127.0.0.1'
10     os.environ['MASTER_PORT'] = '29500'
11     dist.init_process_group(backend, rank=rank, world_size=size)
12     fn(rank, size)
13
14 if __name__ == "__main__":
15     processes = []
16     for rank in range(5):
17         p = Process(target=init_process, args=(rank, size, run))
18         p.start()
19         processes.append(p)
20     for p in processes:
21         p.join()
```

- `torch.distributed` extends into multiple machines/clusters

P2P: Synchronous Communication

Carnegie Mellon



torch_dist_send_recv_sync

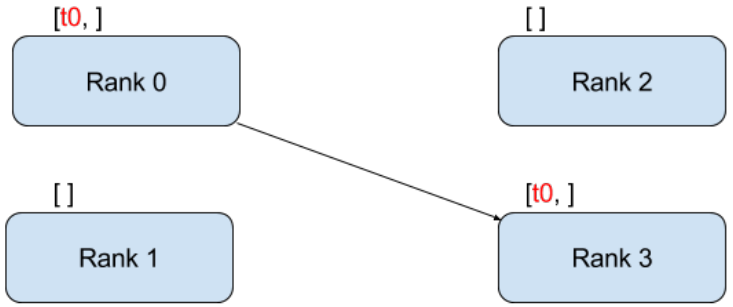
```
1  def run(rank, size):
2      tensor = torch.zeros(1)
3      if rank == 0:
4          tensor += 1
5          dist.send(tensor=tensor, dst=1) # Send the tensor to process 1
6      else:
7          dist.recv(tensor=tensor, src=0) # Receive tensor from process 0
8      print('Rank ', rank, ' has data ', tensor[0])
```

- Processes wait until communication ends to resume

Image credit: Séb Arnold

P2P: Asynchronous Communication

Carnegie Mellon



torch_dist_send_recv_Async

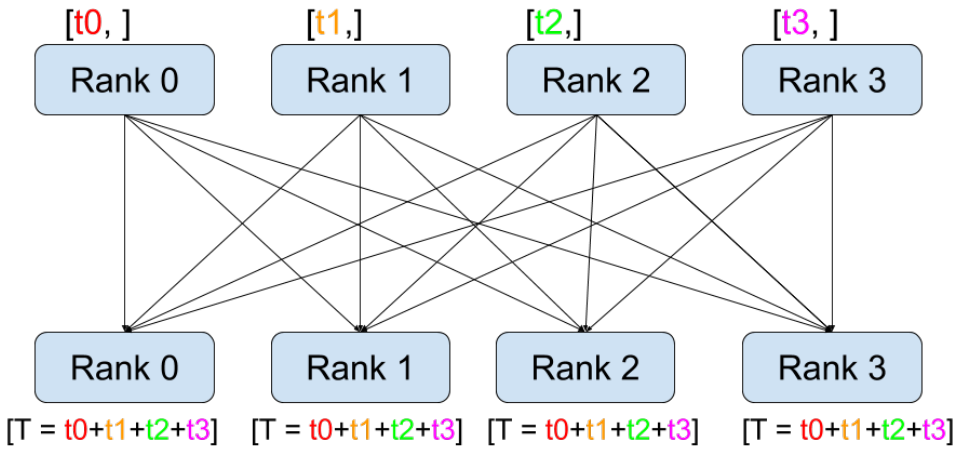
```
1  def run(rank, size):
2      tensor = torch.zeros(1)
3      req = None
4      if rank == 0:
5          tensor += 1
6          req = dist.isend(tensor=tensor, dst=1) # Send the tensor to process 1
7      else:
8          req = dist.irecv(tensor=tensor, src=0) # Receive tensor from process 0
9      req.wait()
10     print('Rank ', rank, ' has data ', tensor[0])
```

- Processes does not have to wait, might cause inconsistencies

Image credit: Séb Arnold

Collective: All Reduce

Carnegie Mellon



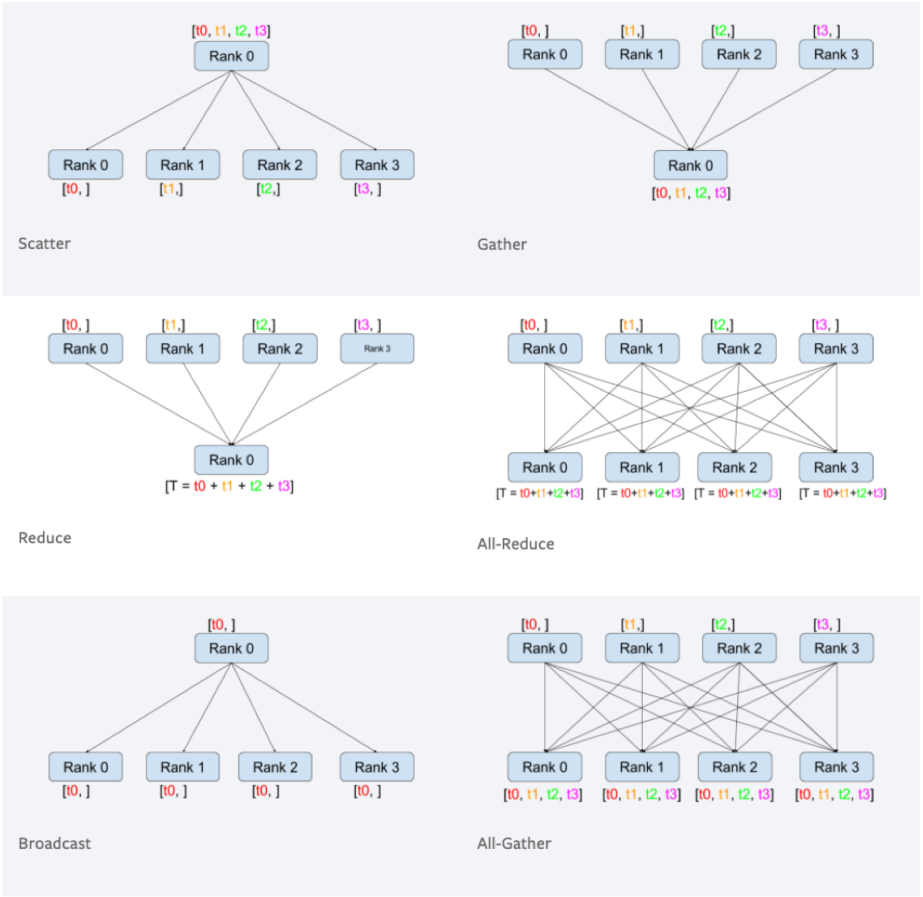
torch_dist_allreduce

```
1  """ Gradient averaging. """
2  def average_gradients(model):
3      size = float(dist.get_world_size())
4      for param in model.parameters():
5          dist.all_reduce(param.grad.data, op=dist.reduce_op.SUM)
6          param.grad.data /= size
```

- All to all with reduce ops: `SUM`, `PRODUCT`, `MAX`, `MIN`

Image credit: Séb Arnold

Collective: Others



- Also can also use a checkpoint/barrier: `dist.barrier()`

Image credit: Séb Arnold

Example: Distributed SGD[8]

Carnegie Mellon

torch_dist_SGD

```
1  def run(rank, size):
2      torch.manual_seed(1234)
3      train_set, bsz = partition_dataset()
4      model = Net()
5      optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
6
7      num_batches = ceil(len(train_set.dataset) / float(bsz))
8      for epoch in range(10):
9          epoch_loss = 0.0
10         for data, target in train_set:
11             optimizer.zero_grad()
12             output = model(data)
13             loss = F.nll_loss(output, target)
14             epoch_loss += loss.item()
15             loss.backward()
16             average_gradients(model)
17             optimizer.step()
18         print('Rank ', dist.get_rank(), ', epoch ', epoch, ': ', epoch_loss / num_batches)
```

- `torch.dist` helps distribute data to backend nodes/ranks

[8] https://pytorch.org/tutorials/intermediate/dist_tuto.html

Distributed Backends[9]

Carnegie Mellon

Backend	gloo		mpi		nccl	
Device	CPU	GPU	CPU	GPU	CPU	GPU
send	✓	✘	✓	?	✘	✘
recv	✓	✘	✓	?	✘	✘
broadcast	✓	✓	✓	?	✘	✓
all_reduce	✓	✓	✓	?	✘	✓
reduce	✓	✘	✓	?	✘	✓
all_gather	✓	✘	✓	?	✘	✓
gather	✓	✘	✓	?	✘	✘
scatter	✓	✘	✓	?	✘	✘
barrier	✓	✘	✓	?	✘	✓

- `gloo`: CPU and some (not optimized for) GPUs (Facebook)
- `ncll`: optimized for CUDA/GPU (NVIDIA)
- `mpi`: support popular platforms: Open-MPI, MVAPICH2, Intel MPI. Needs extra work to get it installed.

[9] `https://pytorch.org/docs/stable/distributed.html`