Distributed PyTorch Training Documentation

This documentation covers a complete distributed PyTorch training setup using SLURM across multiple nodes with both NCCL and Gloo communication backends.

Overview

The setup consists of:

- **6 nodes**: master-node, node01, node02, node03, node04, node05
- **1 GPU per node**: Single-GPU distributed training
- **Multiple backend support**: NCCL (recommended for GPUs) and Gloo (CPU/mixed environments)
- **SLURM orchestration**: Automated job scheduling and process management

Core Components

1. Python Training Script (`py_exe.py`)

The main training script implements PyTorch's Distributed Data Parallel (DDP) pattern:

Key Functions

`setup(rank, world_size, master_addr, master_port)`

- Initializes the distributed process group
- Sets environment variables for master node communication
- Uses NCCL backend for optimal GPU communication

`cleanup()`

- Properly destroys the process group after training
- Essential for clean shutdown and resource cleanup

`SimpleModel`

- Basic neural network with 2 hidden layers (10→50→2)
- Demonstrates typical model structure for distributed training

`main()`

- Orchestrates the entire distributed training process
- Handles device assignment, model wrapping, and training loop

Critical Implementation Details

```python
# Proper device assignment using local rank
local_rank = int(os.environ.get('SLURM_LOCALID', 0))
torch.cuda.set_device(local_rank)
device = torch.device('cuda', local_rank)

# DDP model wrapping
ddp_model = DDP(model, device_ids=[local_rank])

# Distributed data sampling
sampler = DistributedSampler(dataset, num_replicas=world_size, rank=rank)
```

2. NCCL Configuration (`run_nccl.slurm`)

NCCL (NVIDIA Collective Communications Library) is optimized for GPU-to-GPU communication.

SLURM Parameters

- `--nodes=6`: Uses 6 compute nodes
- `--ntasks-per-node=1`: One process per node
- `--gpus-per-node=1`: Single GPU per node
- `--nodelist`: Explicit node specification for consistent assignment

Network Interface Configuration

Each node uses different Ethernet interfaces due to hardware variations:

```bash
case $RANK in
  0) export NCCL_SOCKET_IFNAME=ens1f1 ;;      # master-node
  1) export NCCL_SOCKET_IFNAME=ens1f1 ;;      # node01
  2) export NCCL_SOCKET_IFNAME=ens1f0np0 ;;   # node02
  3) export NCCL_SOCKET_IFNAME=ens1f1 ;;      # node03
  4) export NCCL_SOCKET_IFNAME=ens1f0 ;;      # node04
  5) export NCCL_SOCKET_IFNAME=ens1f0np0 ;;   # node05
esac
```

Key NCCL Settings

- `NCCL_IB_DISABLE=1`: Disables InfiniBand, forces Ethernet usage
- `NCCL_DEBUG=INFO`: Enables diagnostic output for troubleshooting
- `NCCL_SOCKET_IFNAME`: Specifies network interface per node

3. Gloo Configuration (`run_gloo.slurm`)

Gloo backend provides CPU-based communication, useful for mixed CPU/GPU environments or when NCCL isn't available.

Key Differences from NCCL

- Uses `GLOO_SOCKET_IFNAME` instead of `NCCL_SOCKET_IFNAME`
- References `train_gloo.py` (should be same as `py_exe.py` with backend="gloo")
- More flexible for heterogeneous hardware setups

Network Architecture

Master Node Setup

- **IP**: 192.168.20.15
- **Port**: 29500
- **Role**: Coordinates all distributed operations
- **Interface**: ens1f1

Node Communication Flow

1. All nodes connect to master node at initialization
2. SLURM assigns unique `RANK` (0-5) and `WORLD_SIZE` (6) to each process
3. PyTorch DDP handles gradient synchronization across all nodes
4. Each epoch, data is distributed via `DistributedSampler`

Environment Variables

SLURM-Provided

- `SLURM_PROCID`: Process ID (becomes RANK)
- `SLURM_NTASKS`: Total number of tasks (becomes WORLD_SIZE)
- `SLURM_LOCALID`: Local rank within node (for GPU assignment)

User-Defined

- `MASTER_ADDR`: Master node IP address
- `MASTER_PORT`: Communication port
- `NCCL_SOCKET_IFNAME` / `GLOO_SOCKET_IFNAME`: Network interface per node

Usage Instructions

Running NCCL Version

```bash
sbatch run_nccl.slurm
```

Running Gloo Version

```bash
sbatch run_gloo.slurm
```

Monitoring Jobs

```bash
# Check job status
squeue -u $USER

# View output logs
tail -f nccl_<job_id>.out
tail -f nccl_<job_id>.err

# View Gloo logs
tail -f gloo_<job_id>.out
tail -f gloo_<job_id>.err
```

Troubleshooting

Common Issues

1. Network Interface Problems

- Symptom: "Network unreachable" or timeout errors
- Solution: Verify correct interface names with `ip addr show` on each node
- Update `NCCL_SOCKET_IFNAME` or `GLOO_SOCKET_IFNAME` accordingly

2. GPU Allocation Issues

- Symptom: CUDA out of memory or device assignment errors
- Solution: Ensure `SLURM_LOCALID` correctly maps to available GPUs
- Check GPU availability with `nvidia-smi`

3. Process Group Initialization Failures

- Symptom: Timeout during `init_process_group`
- Solution: Verify master node accessibility and firewall settings

- Check that MASTER_ADDR is reachable from all nodes

4. InfiniBand Conflicts

- Symptom: NCCL initialization hangs
- Solution: Ensure `NCCL_IB_DISABLE=1` is set
- Consider using Gloo backend if issues persist

Debugging Tips

1. **Enable Verbose Logging**

```bash
export NCCL_DEBUG=INFO
export PYTHONUNBUFFERED=1
```

2. **Test Network Connectivity**

```bash
# From each node, test master connectivity
ping 192.168.20.15
telnet 192.168.20.15 29500
```

3. **Verify GPU Availability**

```bash
srun --nodes=6 --ntasks-per-node=1 nvidia-smi
```

Performance Considerations

NCCL vs Gloo

- **NCCL**: Faster for GPU-intensive workloads, optimized for NVIDIA hardware
- **Gloo**: More flexible, works with CPU-only setups, better for mixed environments

Scaling Recommendations

- **Batch Size**: Scale linearly with number of nodes (64 × 6 = 384 effective batch size)
- **Learning Rate**: Consider scaling learning rate with batch size
- **Gradient Accumulation**: Use when memory constraints limit batch size per GPU

Network Optimization

- Use dedicated high-bandwidth interfaces when available
- Consider network topology when scaling beyond current setup

- Monitor network utilization during training

Configuration Files Summary

File	Purpose	Backend
`py_exe.py`	Main training script	NCCL (configurable)
`run_nccl.slurm`	SLURM job script for NCCL	NCCL
`run_gloo.slurm`	SLURM job script for Gloo	Gloo

This setup provides a robust foundation for distributed PyTorch training that can scale to larger clusters while maintaining reliable communication across nodes.