Training and Deploying GCN in CrypTen

Anmol Agarwal: 2018CS10327

December 4, 2021

Porting to Newer Version

Since the library of CrypTen is new and still under development phase, it is continually evolving. Hence, there were issues in running the older code on GCN with newer CrypTen library. The issues are as follows:

- **ConfigManager** Library used to decide accuracy and precision of floating point numbers was modified to be used through **crypten.config**.
- Exploding loss values occurred during training. After investigation, following was observed:
  - Bug when multiplying zero tensors for parties ≥ 3. Issue about this was raised on GitHub and resolved. Code shown on Listing 1. Although, after resolving bug, exploding loss values still occurred.
  - Exploding loss values never occurred for 1 party but always occurred for greater than 1 party.
  - The code was running correctly on older CrypTen.
  - Lowering the learning rate lead to delay in explosion of loss values, but they still occurred. Although if learning rate was the reason behind explosion of loss values, then exploding loss values must have been observed for 1 party setting as well.

So as explained above, code was ported to newer version of CrypTen, but it was still difficult to figure out why explosions were happening.

```
import torch
import numpy as np
import crypten
crypten.init()
torch.set_num_threads(1)
import crypten.mpc as mpc
```

```
import crypten.communicator as comm
from crypten.communicator.communicator import _logging

zero_array = np.zeros((5,5))

@mpc.run_multiprocess(world_size=5)
def check_zero():
    rank = comm.get().get_rank()
    zero1 = crypten.cryptensor(zero_array[0],src=0)
    zero2 = crypten.cryptensor(zero_array[0],src=0)
    zero3 = zero1*zero2
    crypten.print(zero3.get_plain_text())
check_zero()

# Output
# tensor([-4.2950e+09,  1.5259e-05,  0.0000e+00,  1.5259e-05,
    -4.2950e+09])
```

Listing 1: Zero Tensor Multiplication Bug

Two Machine Training

CrypTen was earlier trained on a single machine using multiple processes in which case we just needed to specify the world size. But approach was different when training on multiple machines. We needed to specify the rank of each machine, the world size (i.e. number of parties), IP address/port of the master node (rank 0) node, and the backend (Among MPI, Gloo and NCCL, Gloo was used by default). These values were saved in environment variables on each machine and then we could run the code as before.

Measurements

Training measurements were carried out to check communication cost (using tcpdump) in terms of packets received, bytes transferred, time captured and bandwidth, and computation cost (using top command) in terms of CPU and memory overhead.
The machines had the following configuration:

- Machine 1: Intel Core i7 10750H, 16GB RAM
- Machine 2: Intel Core i5 6200U, 8GB RAM

Average ping latency between two machines was 7.9 ms.

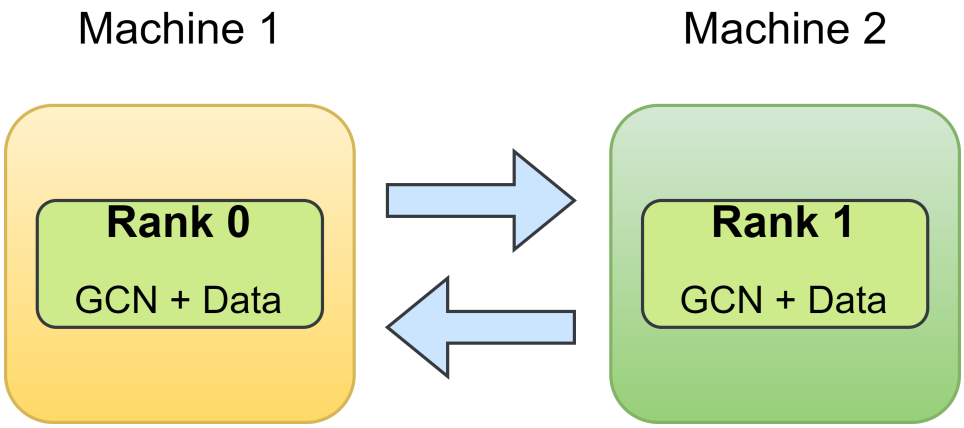


Figure 1: Training on 2 machines.

Statistics

When training, statistics were measured on machine 1 only, first as a rank 0 node and then as a rank 1 node.

Rank	Pkts	Data(MB)	Time/Epoch(s)	Captured(s)	BW(MB/s)	CPU(%)	Mem(%)
0	1002480	1685.89	4.14	844.239	1.997	39.61	1.98
1	1015286	1683.56	4.13	835.239	2.016	41.19	1.90

Table 1: Computation & Communication overhead at rank 0 & 1

Training on AWS Nodes

The scripts were now ready to extend to n-party setting for any value of n. But the approach to run on AWS nodes was slightly different from what we have seen so far, because it was not possible to configure environment variables on each host manually. To resolve this, we used the AWS Launcher Script provided by CrypTen on GitHub.
Note: EC2 instances used were *Deep Learning AMI (Ubuntu 18.04) Version 52.0*

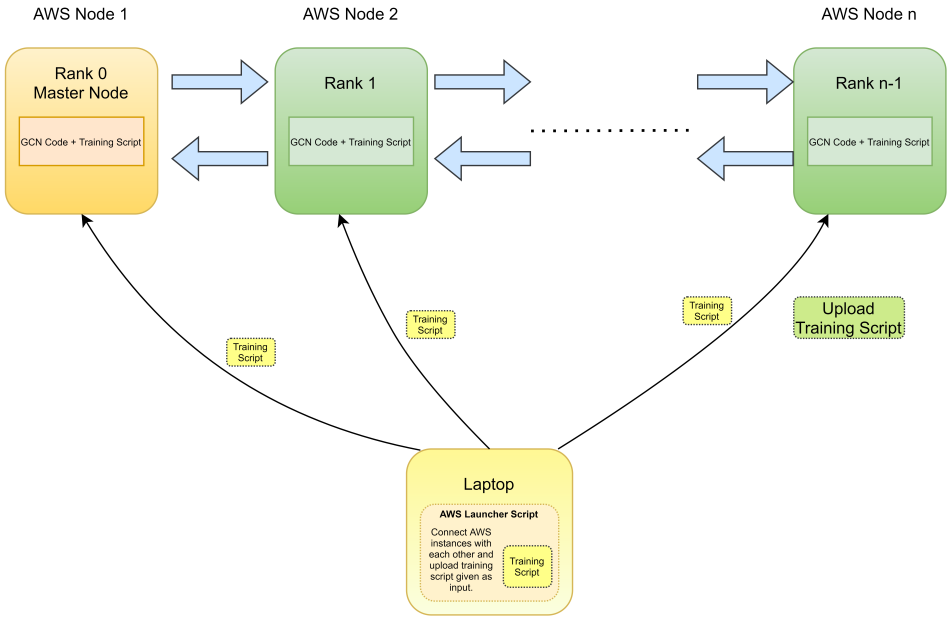


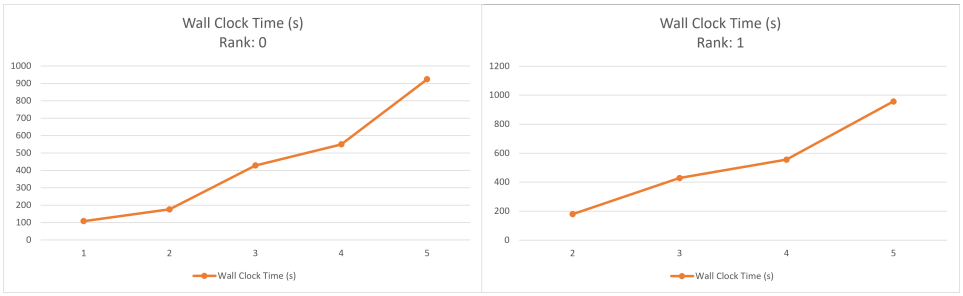
Figure 2: Training on multiple AWS nodes.

Challenges

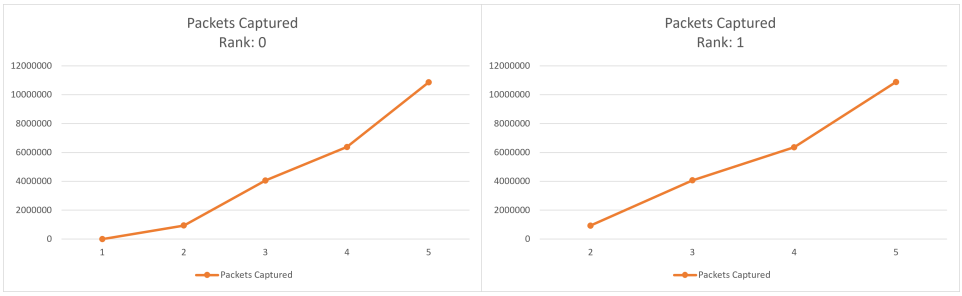
- AWS Nodes were not connecting to each other. Resolved by changing security group rule to accept incoming packets on all ports.
- Needed to create shell script to set up conda environment, and then run the program with its arguments.

Measurements

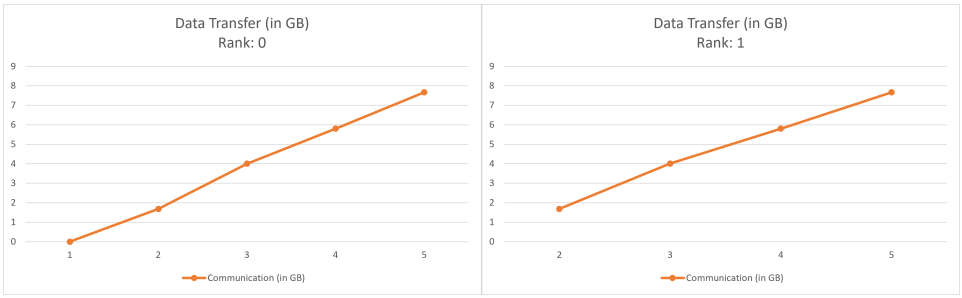
Finally after launching the scripts, the communication and computation costs were measured in a similar manner as with 2 machines, shown in tables (2, 3) and figures (3, 4). As we can see from the figures the load distribution among all nodes is almost equal because rank 0 (master node) and rank 1 (non master node) nodes have same overhead.



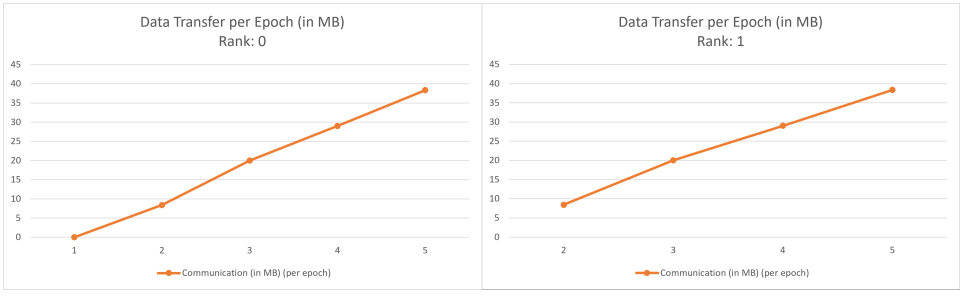
(a) Wall Clock Time



(b) Packets Transferred

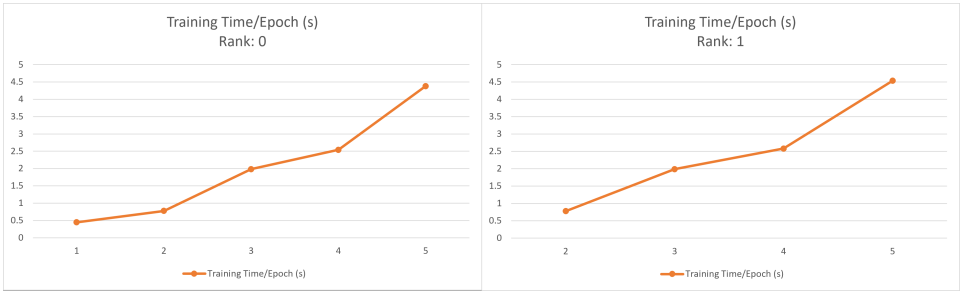


(c) Data Transferred

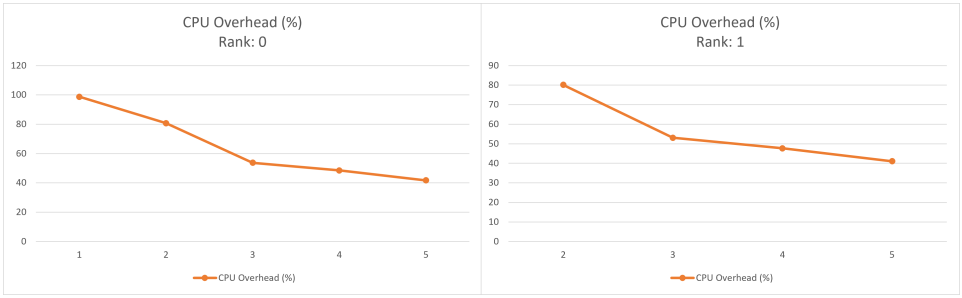


(d) Data Transfer per Epoch

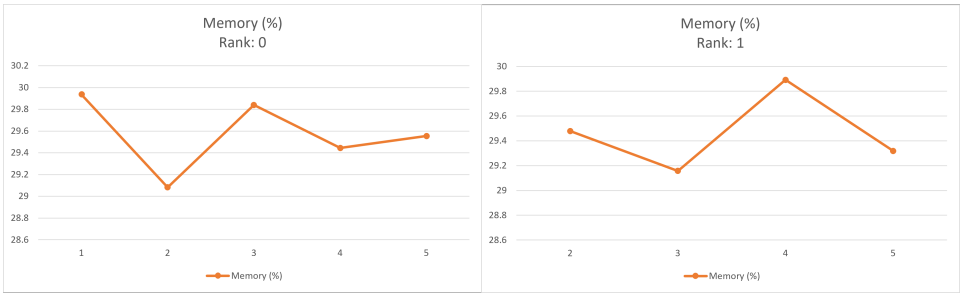
Figure 3



(a) Training Time per Epoch



(b) CPU Overhead (%)



(c) Memory Overhead (%)

Figure 4

Parties	Wall Clk(s)	Pkts	Data(GB)	Data/Epoch(MB)	Time/Epoch(s)	CPU(%)	Mem(%)
1	108.29	0	0	0	0.45	98.70	29.94
2	176	939229	1.68	8.40	0.78	80.76	29.08
3	427.93	4054223	4.0	20.0	1.98	53.74	29.84
4	549.3	6383688	5.80	29.01	2.54	48.52	29.45
5	923.36	10856337	7.67	38.34	4.38	41.74	29.55

Table 2: Communication & Computation Overhead at rank 0

Parties	Wall Clk(s)	Pkts	Data(GB)	Data/Epoch(MB)	Time/Epoch(s)	CPU(%)	Mem(%)
2	179.36	935784	1.68	8.43	0.78	80.1	29.48
3	427.92	4066562	4.01	20.04	1.99	53.07	29.16
4	555.2	6364673	5.80	29.01	2.58	47.68	29.89
5	956.17	10881596	7.67	38.35	4.54	41.07	29.32

Table 3: Communication & Computation Overhead at rank 1

Inferences

- Communication cost in terms of total data transfer, data transfer per epoch and packets captured is linear with number of parties.
- CPU Overhead decreases with more parties, which may be because program is becoming more I/O bound due to higher communication overhead.
- Wall clock time, training time per epoch and packets captured seem to follow a more exponential trend.
- Memory overhead is almost constant for any number of parties, and it was around 320 MB.

Model and Dataset Stats

We had 24 hours of data, out of which we trained on 1 hour of data.

Size

- 1 Hour Dataset Size: 0.14 MB
- PyTorch GCN Model Size: 2.3 kB
- CrypTen GCN Model Size: 2.6 kB

Accuracy

We used RMSE score to measure the accuracy of the GCN PyTorch model.

Epochs	Train RMSE	Test RMSE	Train Time(s)
100	109.95	49.60	0.438
Full Training	40.211	50.80	4.22

Table 4: RMSE Score and Training Time for GCN PyTorch Model

Models	TRAIN_RMSE$_{100}$	TEST_RMSE$_{100}$	TRAIN_TIME$_{100}$	TRAIN_RMSE$_{FULL}$	TEST_RMSE$_{FULL}$	TRAIN_TIME$_{FULL}$	VAR
GPR	29.76	30.41	1176.45	29.13	29.74	3433.04	✓
Variational GPR	32.42	33.05	267.89	29.89	30.63	1053.88	✓
ANN	47.04	47.51	51.78	31.61	32.49	141.42	X
GraphSAGE	X	35.45	4654.64	X	34.15	4891.81	X
Meaner	X	X	X	X	33.84	1578.64	X
GCN	109.95	49.60	0.438	40.211	50.80	4.22	X

Table 5: RMSE Score Comparison with other models

- Full Training means training the model until it converges, hence the number of epochs to train varies over different models.
- Since, we couldn't resolve exploding loss values in CrypTen, we weren't able to take measurements for GCN CrypTen model, but ideally the results should be the same.

Future Scope

Larger Models

As we can see in table 5 that even if GCN has higher RMSE values, it has very low training time compared to other models. Moreover, there were only two GCN layers in the model and its size was very small (2.3kB). Hence we can extend to bigger and more sophisticated models based on GCN which may result in lower RMSE values and low training time.

Communication Overhead

Communication and memory overhead is huge in CrypTen on each node for model and dataset of size in kBs. Hence we may need to take care of that when implementing more complex models in future.