

## Mansoor Nabawi, 309498

---

DDA Tutorial 09, 07.10.2022


I am using Google Colab for this exercise and as it is free it only has 2 cores.

First, we install mpi4py in Colab using the following code

```
- !pip install mpi4py
```

For this exercise as one python file was so long, I created multiple python files and insert them whenever needed, so I reduce the lines of code in the final python file as well as the readability, which will be better.

The first python file I created is the **net.py** which contains my CNN architecture. Here is my architecture summary.

 `%run net.py`

```
print(Net())
```

```
Net(  
  (dropout): Dropout2d(p=0.2, inplace=False)  
  (conv1): Conv2d(1, 8, kernel_size=(5, 5), stride=(1, 1))  
  (conv2): Conv2d(8, 16, kernel_size=(5, 5), stride=(1, 1))  
  (relu): ReLU()  
  (fc1): Linear(in_features=64, out_features=120, bias=True)  
  (fc2): Linear(in_features=120, out_features=10, bias=True)  
)
```

Next, I wrote all the data loading and transformation into another python file called **data\_load.py**

In this file, I am transforming the MNIST data to tensors and normalizing them as well. (I used my code from the previous exercise, therefore I don't put a screenshot as it is an easy and simple task.)

And my final python file is called **parallel\_sgd.py**.

In this code first, I load the model architecture from the file called net and as we are using mpi4py I also import it plus other needed libraries.

After importing the libraries we need to initialize the mpi as well as start to time the run of our code.

As always we need to initialize comm, rank, and size. We can do them by the following code.

```
comm = MPI.COMM_WORLD  
rank = comm.Get_rank()  
size = comm.Get_size()
```

Next, I defined a training loop inside of a function.

In this function as we may use more than one processor, it is important to know how the data is distributed so first print the size of the data and the rank(processor) in our training is happening.

As always we go through the epochs

- and then in each epoch
  - We look into each batch,
  - We get the data in each batch
  - And performing the training
  - We get the loss and accuracy in each epoch and
  - If the difference of the loss of this epoch is less than  $10e-5$  from the loss of the previous epoch.
    - Then it is converged.
    - Break the loop

```
#defining a training function loop.
def train(model, optimizer, loss_fn, dataloader, epochs, rank=0):
    size = len(dataloader.dataset)

    print(f'Training started...\n{size} data samples ---> Worker: {rank}')

    for epoch in range(epochs):
        loss_total = 0
        accuracy = 0
        for _, (X, y) in enumerate(dataloader):

            optimizer.zero_grad()
            #prediction
            y_pred = model(X)
            #loss
            loss = loss_fn(y_pred, y)

            # Backpropagation
            loss.backward()
            optimizer.step()
            loss_total += loss.item()*X.size(0)

            # Track the accuracy
            accuracy += (y_pred.argmax(1) == y).type(torch.float).sum().item()

        loss_total /= size
        accuracy /= size
        print(f'rank: {rank} -- epoch: {epoch} --> loss: {loss_total:>4f} -- accuracy: {(100*accuracy):>0.1f}%')
        if epoch!=0:
            if last_loss - loss_total < 10e-5:
                print(f'--- Converged on rank {rank} after {epoch} epochs ---')
                break

        last_loss = loss_total
    return
```

If we have only one processor, we just import the data and train using the full data.

If we have more than one processor then we need to split the data, we split the data using `random_split()` of torch library. For example, if we have 4 processors we will have data samples of size 15000 for each rank. Here is the code.

```

#if there is more than one processor
if rank==0:
    #loading data
    from data_load import *

    if size>1:
        #splitting data
        len_datasets = [len(train_data.targets)//size]*size
        train_data = [ii for ii in torch.utils.data.random_split(train_data,len_datasets)]

else:
    train_data = None

```

We use scatter to distribute data evenly between processes. We then use dataloader to make batches of our dataset. We make an instance of our model and save the model parameters if we are in rank 0. As well as we broadcast these parameters from rank 0. Then in other processes, we use these parameters to update our model parameters and train our model.

```

state_dict = {}
if rank==0:
    for name, param in model.named_parameters():
        state_dict[name] = param.detach().numpy()

state_dicts = comm.bcast(state_dict, root=0)

if rank!=0:
    for name, param in model.named_parameters():
        model.state_dict()[name][:] = torch.Tensor(state_dicts[name])

```

We use CrossEntropyLoss as loss, and we use SGD as our optimizer with the new parameters and a learning rate 0.1.

We train our model in each using the selected optimizer and loss and on a specific rank, and in 20 epochs.

```

loss_fn = nn.CrossEntropyLoss()

optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

train(model,optimizer,loss_fn,train_iterator,epochs=20,rank=rank)

state_dict = {}
for name, param in model.named_parameters():
    state_dict[name] = param.detach().numpy()

```

We then save the parameters. And gather these parameters. And finally, save the time.

The results are as below. Because of the long runtime, I used accuracy in the training set instead of running a test set even though it is not a hard thing to write a function for testing dataset.

```
[ ] !mpirun --allow-run-as-root -np 1 python parallel_sgd.py
```

```
Training started...
```

```
60000 data samples ---> Worker: 0
```

```
rank: 0 -- epoch: 1 --> loss: 2.051886 -- accuracy: 41.5%
```

```
rank: 0 -- epoch: 2 --> loss: 1.611036 -- accuracy: 86.0%
```

```
rank: 0 -- epoch: 3 --> loss: 1.535477 -- accuracy: 93.0%
```

```
rank: 0 -- epoch: 4 --> loss: 1.520393 -- accuracy: 94.3%
```

```
rank: 0 -- epoch: 5 --> loss: 1.512629 -- accuracy: 95.1%
```

```
rank: 0 -- epoch: 6 --> loss: 1.506944 -- accuracy: 95.6%
```

```
rank: 0 -- epoch: 7 --> loss: 1.503842 -- accuracy: 95.9%
```

```
rank: 0 -- epoch: 8 --> loss: 1.501136 -- accuracy: 96.1%
```

```
rank: 0 -- epoch: 9 --> loss: 1.498427 -- accuracy: 96.4%
```

```
rank: 0 -- epoch: 10 --> loss: 1.496949 -- accuracy: 96.6%
```

```
rank: 0 -- epoch: 11 --> loss: 1.494980 -- accuracy: 96.7%
```

```
rank: 0 -- epoch: 12 --> loss: 1.493508 -- accuracy: 96.9%
```

```
rank: 0 -- epoch: 13 --> loss: 1.492115 -- accuracy: 97.0%
```

```
rank: 0 -- epoch: 14 --> loss: 1.491535 -- accuracy: 97.1%
```

```
rank: 0 -- epoch: 15 --> loss: 1.490801 -- accuracy: 97.1%
```

```
rank: 0 -- epoch: 16 --> loss: 1.489415 -- accuracy: 97.2%
```

```
rank: 0 -- epoch: 17 --> loss: 1.488305 -- accuracy: 97.4%
```

```
rank: 0 -- epoch: 18 --> loss: 1.486934 -- accuracy: 97.5%
```

```
rank: 0 -- epoch: 19 --> loss: 1.486696 -- accuracy: 97.6%
```

```
rank: 0 -- epoch: 20 --> loss: 1.486246 -- accuracy: 97.6%
```

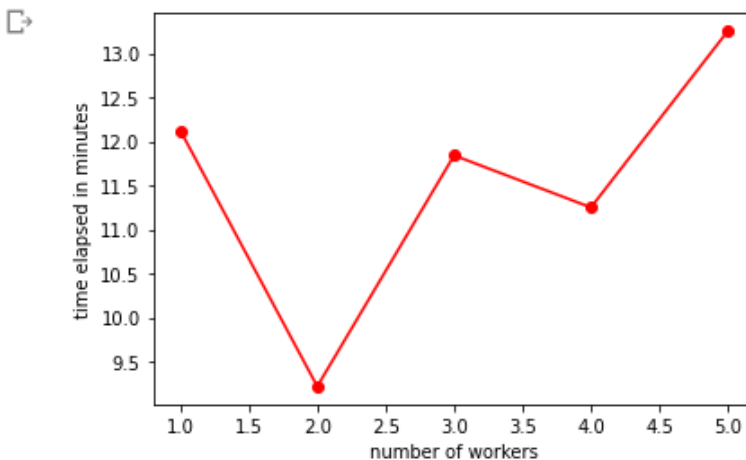
```
Total Time taken by 0 : 12.12 minutes
```

```

import pandas as pd
import matplotlib.pyplot as plt

time_df = pd.DataFrame([[1,12.12],
                        [2,9.22],
                        [3,11.84],
                        [4,11.25],
                        [5,13.25]], columns=["rank", "time_minutes"])
time_df.set_index("rank", inplace=True)
plt.plot(time_df.time_minutes, 'r-o')
plt.xlabel("number of workers")
plt.ylabel("time elapsed in minutes")
plt.show()

```



- Using 1 processor it takes 12 minutes and 12seconds to train with 20 epochs the model did not converge.
- As we increase the number of processors the speed of training is becoming better and it takes less time than before.
- Using 2 processors it takes 9 minutes and 22 seconds to train the whole dataset. The model converged after 18 epochs on rank 1, and rank 0 went full epochs but as we have distributed the dataset the accuracy or loss is not as good as we had on the first try with only one processor. This is the best timing so far.
- Using 3 processors we see the best timing as it takes 11 minutes and 84 seconds. All ranks converged.
- With 4 processors it took 11 minutes and 25 seconds to run the whole parallelized code, all ranks converged except rank 3.
- Increasing to 5 processors we have our worst time performance with 13 minutes and 25 seconds.
- This long runtime is because I am working on colab and I only have 2 CPUs

---

## PyTorch distributed execution (10 points)

Ref: <https://towardsdatascience.com/this-is-hogwild-7cc80cd9b944>

To do this task all the codes are as previous except for the data distribution process.

I am loading data from data\_load.py file and my architecture is loaded from net.py.

The main part of this task is as below.

```
if __name__ == '__main__':

    num_processes = sys.argv[1] #input("import the number of processors: ")
    start = time.time()
    num_processes = int(num_processes)
    model = Net()
    model.share_memory()

    processes = []
    for rank in range(num_processes):
        data_loader = DataLoader(
            dataset=train_data,
            sampler=DistributedSampler(
                dataset=train_data,
                num_replicas=num_processes,
                rank=rank
            ),
            batch_size=32
        )
        p = mp.Process(target=train, args=(model, data_loader, rank))
        p.start()
        processes.append(p)
    for p in processes:
        p.join()

    end = time.time() - start
    print(f'Time taken: {end/60} minutes')
```

When the program runs it gets the number of cpu through terminal arguments.

Here, we instantiate the model and push it to shared memory with the single method call *share\_memory*. I have loaded the dataset in another python file so we only load it at the beginning and here we have it.

We loop over the processes and define a data loader for each process. The data loaders hold a distributed sampler that knows the rank of the process and handles the distribution of the data. Hence, each process has its data partition. The multiprocessing package calls the training function within each process and waits, with the *join* command, for the processes to finish. During training, all the processes have access to the shared model but train only on their very own data partition. Thus, we decrease the training time by approximately 4 which is the number of total training processes.

Let's look at the result.

```
[31] !python pytorch_sgd.py 1
```

```
Training started...
60000 data samples ---> Worker: 0
rank: 0 -- epoch: 1 --> loss: 2.071499 -- accuracy: 39.5%
rank: 0 -- epoch: 2 --> loss: 1.680541 -- accuracy: 78.9%
rank: 0 -- epoch: 3 --> loss: 1.550737 -- accuracy: 91.6%
rank: 0 -- epoch: 4 --> loss: 1.522816 -- accuracy: 94.2%
rank: 0 -- epoch: 5 --> loss: 1.513834 -- accuracy: 95.0%
rank: 0 -- epoch: 6 --> loss: 1.508129 -- accuracy: 95.5%
rank: 0 -- epoch: 7 --> loss: 1.504082 -- accuracy: 95.9%
rank: 0 -- epoch: 8 --> loss: 1.501309 -- accuracy: 96.2%
rank: 0 -- epoch: 9 --> loss: 1.499094 -- accuracy: 96.3%
rank: 0 -- epoch: 10 --> loss: 1.496882 -- accuracy: 96.5%
rank: 0 -- epoch: 11 --> loss: 1.494816 -- accuracy: 96.8%
rank: 0 -- epoch: 12 --> loss: 1.493552 -- accuracy: 96.9%
rank: 0 -- epoch: 13 --> loss: 1.491733 -- accuracy: 97.0%
rank: 0 -- epoch: 14 --> loss: 1.491156 -- accuracy: 97.1%
rank: 0 -- epoch: 15 --> loss: 1.489260 -- accuracy: 97.3%
rank: 0 -- epoch: 16 --> loss: 1.488794 -- accuracy: 97.3%
rank: 0 -- epoch: 17 --> loss: 1.488037 -- accuracy: 97.4%
rank: 0 -- epoch: 18 --> loss: 1.487722 -- accuracy: 97.4%
rank: 0 -- epoch: 19 --> loss: 1.486851 -- accuracy: 97.5%
rank: 0 -- epoch: 20 --> loss: 1.485495 -- accuracy: 97.6%
Time taken: 6.276986932754516 minutes
```

We can see the runtime in this plot and the runtime is reducing the whole time of increasing the number of CPUs. the overall result is way better than our manual sgd and parallelized program.

```
import pandas as pd
import matplotlib.pyplot as plt

time_df = pd.DataFrame([[1,6.27],
                        [2,4.87],
                        [3,4.42],
                        [4,4.19],
                        [5,3.42]], columns=["rank", "time_minutes"])
time_df.set_index("rank", inplace=True)
plt.plot(time_df.time_minutes, 'r-o')
plt.xlabel("number of workers")
plt.ylabel("time elapsed in minutes")
plt.show()
```

