Lab Course: Distributed Data Analytics
Exercise Sheet 9
Group 2 - Monday

Submitted by: Sruthy Annie Santhosh, 312213

- My System

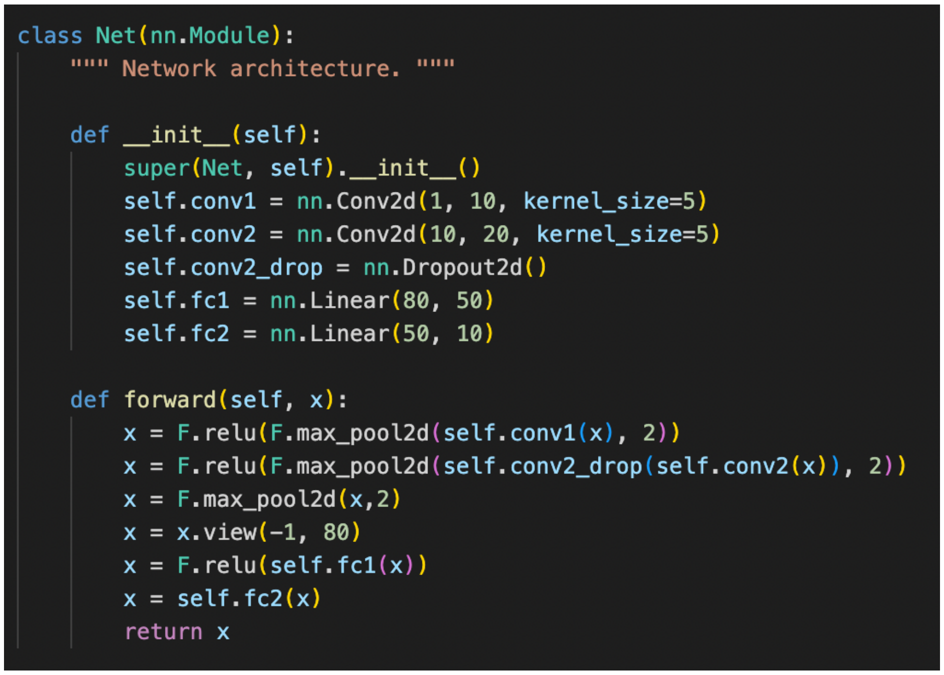
Hardware and Software SetUp:

Processor	M1 chip
Operating System	MacOS Monterey
Number of Cores	8
RAM	8 GB
Python version	3.8.8

The code editor used in this sheet : VScode
The output images attached are screenshots.

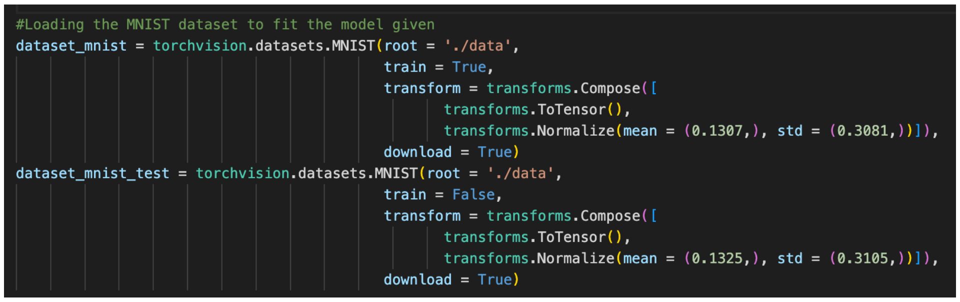
- ## Exercise 1: Implementing Parallel Stochastic Gradient Descent

As given the exercise sheet, I have implemented a CNN model. I have chosen the output channels for convolutional layers randomly. The kernel size is taken as 5 for convolutional layers and 2 for maxpool layers. The stride is always taken as 1 and padding 0. For the fully connected layers the neurons are specified satisfying the formula of the neural network models. Finally, since the dataset used is MNIST, the input channel is 1 and output is 10 classes. I am using cross entropy loss and hence the softmax layer is not used. Dropout layers are also added as specified.



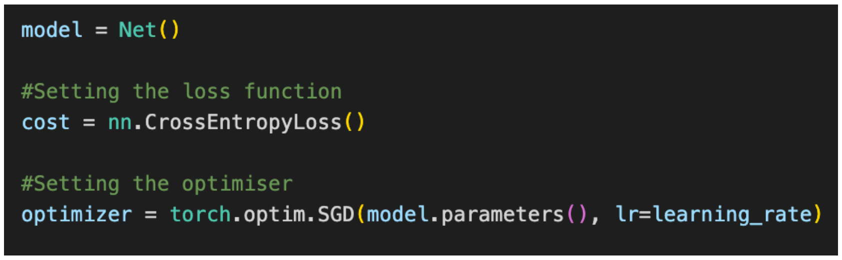
```python
class Net(nn.Module):
    """ Network architecture. """

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(80, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = F.max_pool2d(x,2)
        x = x.view(-1, 80)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

The MNIST dataset is downloaded with data normalisation for train and test set.



```python
#Loading the MNIST dataset to fit the model given
dataset_mnist = torchvision.datasets.MNIST(root = './data',
                                           train = True,
                                           transform = transforms.Compose([
                                               transforms.ToTensor(),
                                               transforms.Normalize(mean = (0.1307,), std = (0.3081,))]),
                                           download = True)
dataset_mnist_test = torchvision.datasets.MNIST(root = './data',
                                           train = False,
                                           transform = transforms.Compose([
                                               transforms.ToTensor(),
                                               transforms.Normalize(mean = (0.1325,), std = (0.3105,))]),
                                           download = True)
```

I have used SGD optimizer and learning rate of 0.01. The batch size is taken as 128 for training and 50 for test and number of epochs 10.

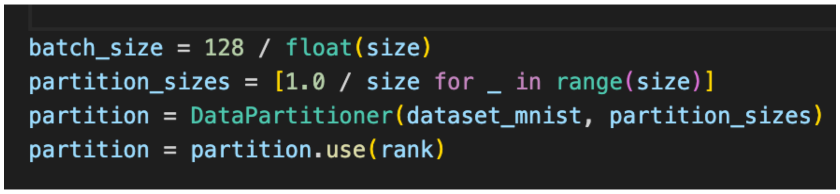


```python
model = Net()

#Setting the loss function
cost = nn.CrossEntropyLoss()

#Setting the optimiser
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

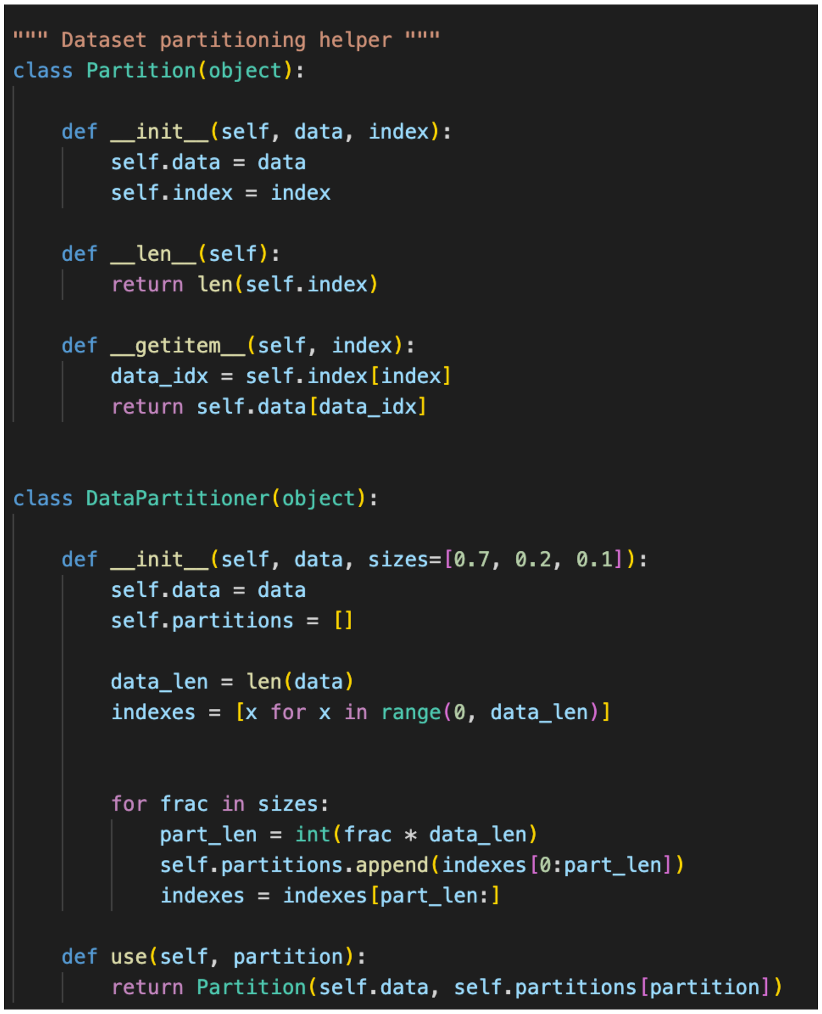
Here we need to do Parallel Stochastic Gradient Descent (PSGD) using mpi on the CNN architecture. The idea behind PSGD is to run SGD on a smaller partition of the whole dataset on P many workers and aggregate the learned weights of the model at the end. We do this using pytorch.

First we need to partition the dataset so that each worker can do the training on one partition.

I have used the classes Partition and DataPartitioner from the pytorch documentation. Here first we get the sizes for each partition and then the datasets and the sizes are passed to the DataPartitioner class. In the class, the sizes are initialized and find the indexes in the dataset for each partition and the use function is called to get the required partition depending on the rank of the process.

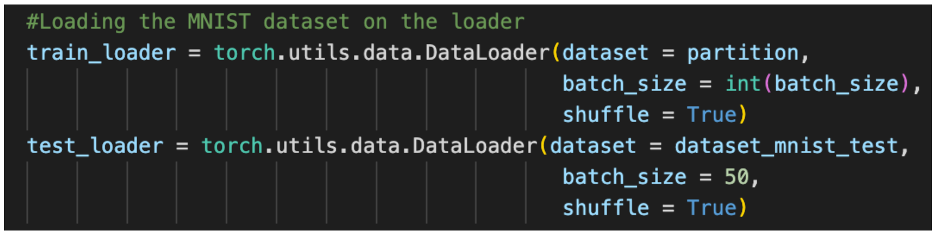


```python
batch_size = 128 / float(size)
partition_sizes = [1.0 / size for _ in range(size)]
partition = DataPartitioner(dataset_mnist, partition_sizes)
partition = partition.use(rank)
```



```python
""" Dataset partitioning helper """
class Partition(object):

    def __init__(self, data, index):
        self.data = data
        self.index = index

    def __len__(self):
        return len(self.index)

    def __getitem__(self, index):
        data_idx = self.index[index]
        return self.data[data_idx]


class DataPartitioner(object):

    def __init__(self, data, sizes=[0.7, 0.2, 0.1]):
        self.data = data
        self.partitions = []

        data_len = len(data)
        indexes = [x for x in range(0, data_len)]


        for frac in sizes:
            part_len = int(frac * data_len)
            self.partitions.append(indexes[0:part_len])
            indexes = indexes[part_len:]

    def use(self, partition):
        return Partition(self.data, self.partitions[partition])
```

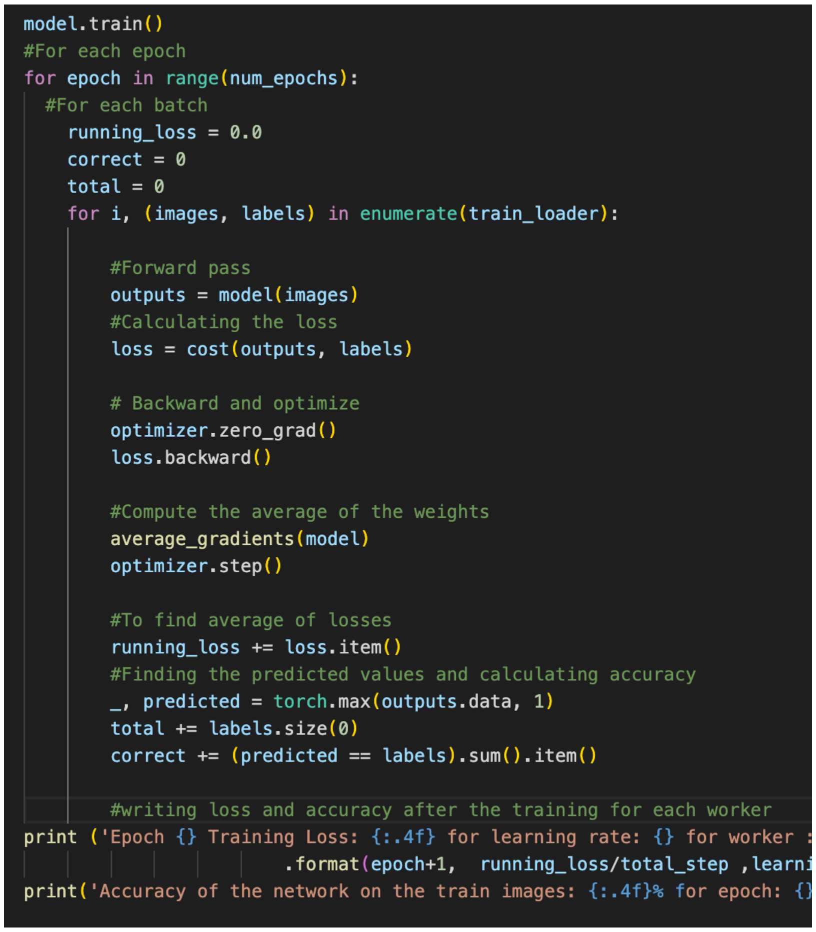
Now the data is loaded according to the partitions for each process.



```python
#Loading the MNIST dataset on the loader
train_loader = torch.utils.data.DataLoader(dataset = partition,
                                           batch_size = int(batch_size),
                                           shuffle = True)
test_loader = torch.utils.data.DataLoader(dataset = dataset_mnist_test,
                                          batch_size = 50,
                                          shuffle = True)
```

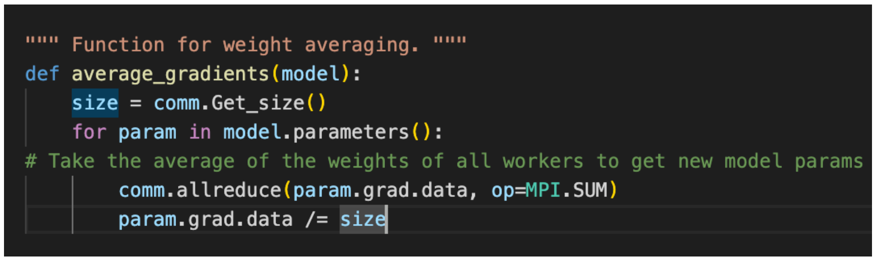
The training is done as follows:

- For each epoch, training is carried out
- Iterating through each batch of images
- The outputs are found and loss is propagated backwards
- The weights are averaged by each process
- The loss is appended and average of loss.The accuracy is found by comparing the predicted values and true values.
- The losses and accuracy are printed at the end. Here the convergence criteria taken is the maximum number of iterations - 10.



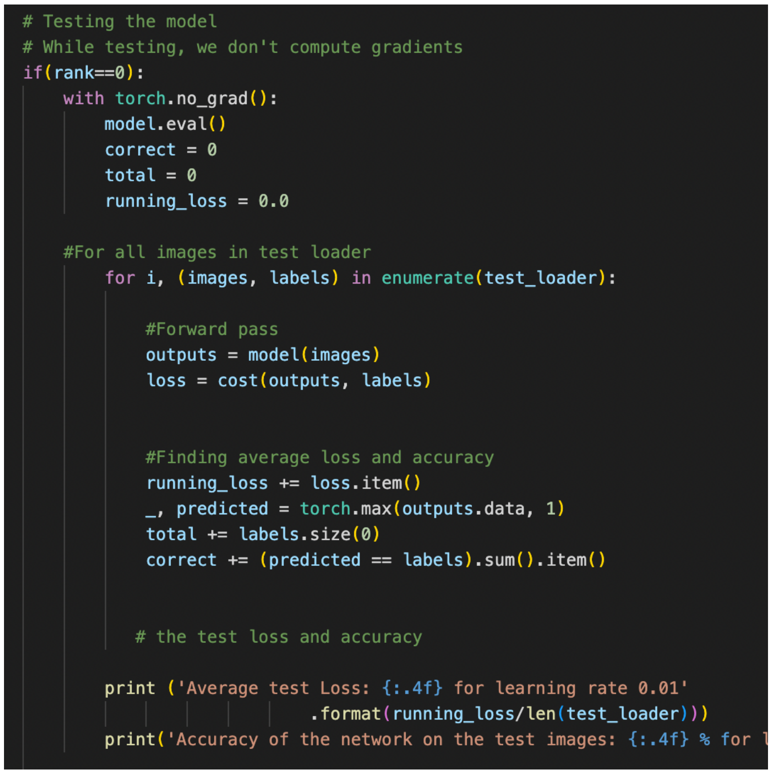
```python
model.train()
#For each epoch
for epoch in range(num_epochs):
  #For each batch
    running_loss = 0.0
    correct = 0
    total = 0
    for i, (images, labels) in enumerate(train_loader):

        #Forward pass
        outputs = model(images)
        #Calculating the loss
        loss = cost(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()

        #Compute the average of the weights
        average_gradients(model)
        optimizer.step()

        #To find average of losses
        running_loss += loss.item()
        #Finding the predicted values and calculating accuracy
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        #writing loss and accuracy after the training for each worker
print ('Epoch {} Training Loss: {:.4f} for learning rate: {} for worker :
                        .format(epoch+1,  running_loss/total_step ,learni
print('Accuracy of the network on the train images: {:.4f}% for epoch: {}
```

-
- The averaging of weights is done by each worker using pytorch. Pytorch allows us to access the parameters of the model by the function model.named_parameters().
- Here I have used comm.allreduce() to gather and find the sum of all the weights and the average is found by dividing by the num of workers.



```
""" Function for weight averaging. """
def average_gradients(model):
    size = comm.Get_size()
    for param in model.parameters():
# Take the average of the weights of all workers to get new model params
        comm.allreduce(param.grad.data, op=MPI.SUM)
        param.grad.data /= size
```

Testing is carried out when the rank==0 i.e, only for the master worker. The model is set to eval mode. For testing gradient is not computed and losses are also not propagated back. The losses for each image batch is appended and mean is taken to get the test loss. The accuracy of the test set is also found out.



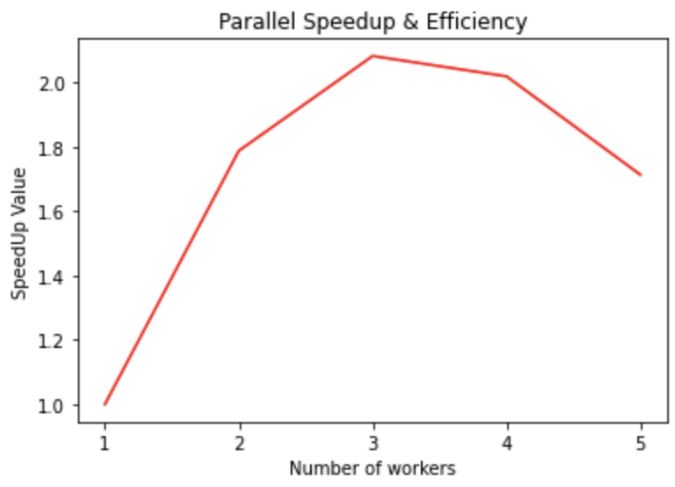
```
# Testing the model
# While testing, we don't compute gradients
if(rank==0):
    with torch.no_grad():
        model.eval()
        correct = 0
        total = 0
        running_loss = 0.0

    #For all images in test loader
        for i, (images, labels) in enumerate(test_loader):

            #Forward pass
            outputs = model(images)
            loss = cost(outputs, labels)


            #Finding average loss and accuracy
            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()


           # the test loss and accuracy

        print ('Average test Loss: {:.4f} for learning rate 0.01'
                    .format(running_loss/len(test_loader)))
        print('Accuracy of the network on the test images: {:.4f} % for l
```

The time is also calculated for the execution of the whole program.
The time taken for running the program for the number of processes, P = 1,2,3,4,5 are recorded as below

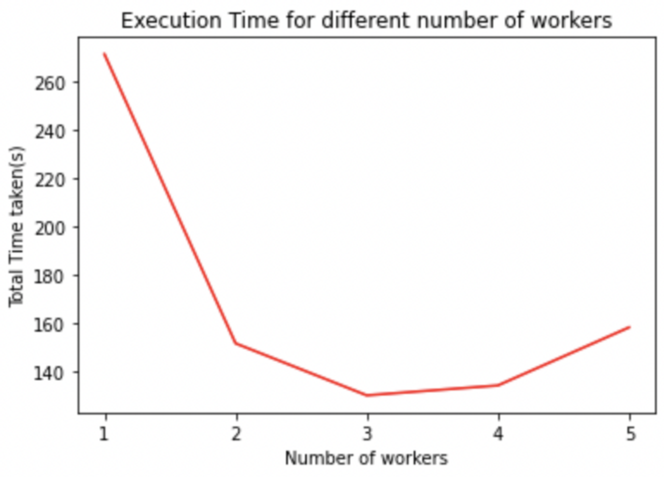
Number of workers	Time in seconds(s)
1	271.3823
2	151.8623

3	130.3759
4	134.4552
5	158.4506

SpeedUp graph



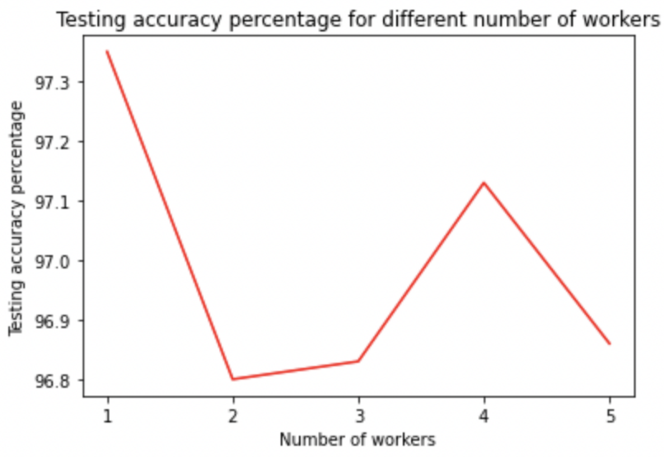
Time taken graph



Observation:

We see that the time decreases steeply initially from P=1 to 2. Then the decrease is more slow. For P=4 and 5, the time taken is increasing slightly. This can be due to the computational overhead of having larger number of processes. But this time is still better than the time taken serially. Hence parallelization reduced the time.

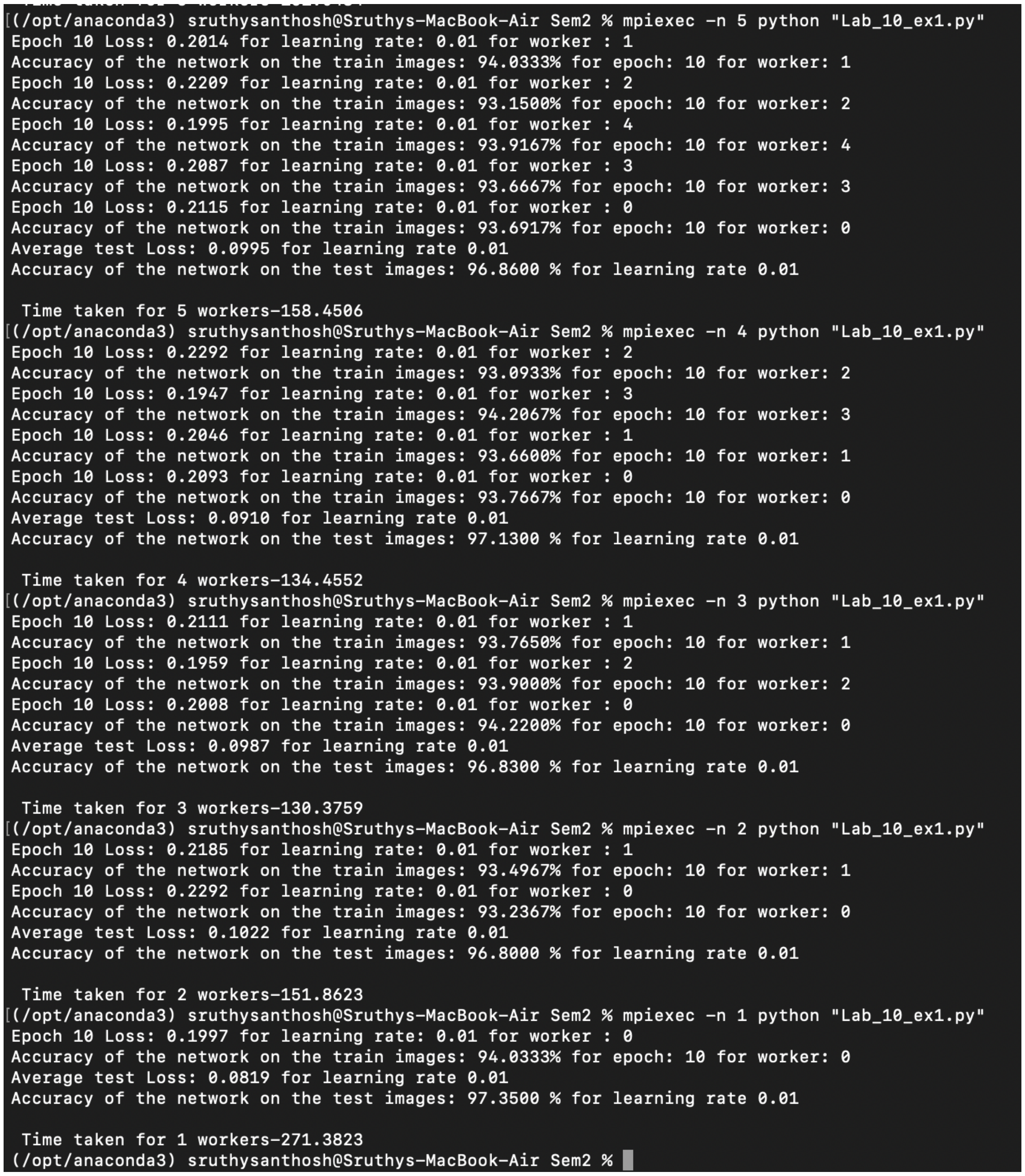
We also plot the testing accuracy of the model for P=1,2,3,4,5 as below



Observation:

We can see that the accuracy is the highest for P=1. And then it increases and decreases. There is no proper trend. And the differences is very less. Hence I don’t think there is any significant relation to accuracy wrt to number of workers used. Hence accuracy is not adversely affected.

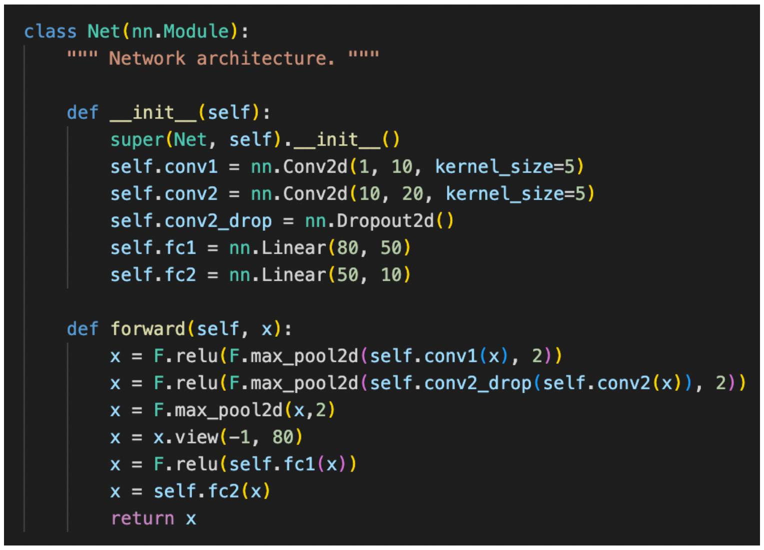
Output Screenshot:


```
[(/opt/anaconda3) sruthysanthosh@Sruthys-MacBook-Air Sem2 % mpiexec -n 5 python "Lab_10_ex1.py"
 Epoch 10 Loss: 0.2014 for learning rate: 0.01 for worker : 1
 Accuracy of the network on the train images: 94.0333% for epoch: 10 for worker: 1
 Epoch 10 Loss: 0.2209 for learning rate: 0.01 for worker : 2
 Accuracy of the network on the train images: 93.1500% for epoch: 10 for worker: 2
 Epoch 10 Loss: 0.1995 for learning rate: 0.01 for worker : 4
 Accuracy of the network on the train images: 93.9167% for epoch: 10 for worker: 4
 Epoch 10 Loss: 0.2087 for learning rate: 0.01 for worker : 3
 Accuracy of the network on the train images: 93.6667% for epoch: 10 for worker: 3
 Epoch 10 Loss: 0.2115 for learning rate: 0.01 for worker : 0
 Accuracy of the network on the train images: 93.6917% for epoch: 10 for worker: 0
 Average test Loss: 0.0995 for learning rate 0.01
 Accuracy of the network on the test images: 96.8600 % for learning rate 0.01

  Time taken for 5 workers-158.4506
[(/opt/anaconda3) sruthysanthosh@Sruthys-MacBook-Air Sem2 % mpiexec -n 4 python "Lab_10_ex1.py"
 Epoch 10 Loss: 0.2292 for learning rate: 0.01 for worker : 2
 Accuracy of the network on the train images: 93.0933% for epoch: 10 for worker: 2
 Epoch 10 Loss: 0.1947 for learning rate: 0.01 for worker : 3
 Accuracy of the network on the train images: 94.2067% for epoch: 10 for worker: 3
 Epoch 10 Loss: 0.2046 for learning rate: 0.01 for worker : 1
 Accuracy of the network on the train images: 93.6600% for epoch: 10 for worker: 1
 Epoch 10 Loss: 0.2093 for learning rate: 0.01 for worker : 0
 Accuracy of the network on the train images: 93.7667% for epoch: 10 for worker: 0
 Average test Loss: 0.0910 for learning rate 0.01
 Accuracy of the network on the test images: 97.1300 % for learning rate 0.01

  Time taken for 4 workers-134.4552
[(/opt/anaconda3) sruthysanthosh@Sruthys-MacBook-Air Sem2 % mpiexec -n 3 python "Lab_10_ex1.py"
 Epoch 10 Loss: 0.2111 for learning rate: 0.01 for worker : 1
 Accuracy of the network on the train images: 93.7650% for epoch: 10 for worker: 1
 Epoch 10 Loss: 0.1959 for learning rate: 0.01 for worker : 2
 Accuracy of the network on the train images: 93.9000% for epoch: 10 for worker: 2
 Epoch 10 Loss: 0.2008 for learning rate: 0.01 for worker : 0
 Accuracy of the network on the train images: 94.2200% for epoch: 10 for worker: 0
 Average test Loss: 0.0987 for learning rate 0.01
 Accuracy of the network on the test images: 96.8300 % for learning rate 0.01

  Time taken for 3 workers-130.3759
[(/opt/anaconda3) sruthysanthosh@Sruthys-MacBook-Air Sem2 % mpiexec -n 2 python "Lab_10_ex1.py"
 Epoch 10 Loss: 0.2185 for learning rate: 0.01 for worker : 1
 Accuracy of the network on the train images: 93.4967% for epoch: 10 for worker: 1
 Epoch 10 Loss: 0.2292 for learning rate: 0.01 for worker : 0
 Accuracy of the network on the train images: 93.2367% for epoch: 10 for worker: 0
 Average test Loss: 0.1022 for learning rate 0.01
 Accuracy of the network on the test images: 96.8000 % for learning rate 0.01

  Time taken for 2 workers-151.8623
[(/opt/anaconda3) sruthysanthosh@Sruthys-MacBook-Air Sem2 % mpiexec -n 1 python "Lab_10_ex1.py"
 Epoch 10 Loss: 0.1997 for learning rate: 0.01 for worker : 0
 Accuracy of the network on the train images: 94.0333% for epoch: 10 for worker: 0
 Average test Loss: 0.0819 for learning rate 0.01
 Accuracy of the network on the test images: 97.3500 % for learning rate 0.01

  Time taken for 1 workers-271.3823
 (/opt/anaconda3) sruthysanthosh@Sruthys-MacBook-Air Sem2 %
```

- **Exercise 2: PyTorch distributed execution**

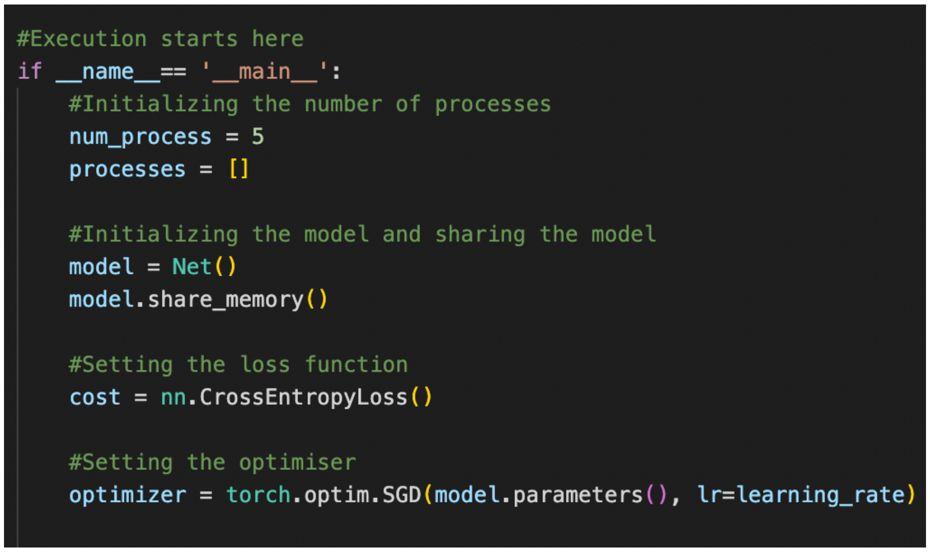
In this exercise, we use PyTorch to handle the multiprocessing nature of the execution. We have to implement HogWild SGD using the torch.multiprocessing library. We will be using a share memory architecture here. We don't need to explicitly take the average of weights here. It is being handled by the torch.multiprocessing library. The total time taken for the execution of the program is also being recorded.

Here also I am using the same CNN model as the previous question.



```
class Net(nn.Module):
    """ Network architecture. """

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(80, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = F.max_pool2d(x,2)
        x = x.view(-1, 80)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

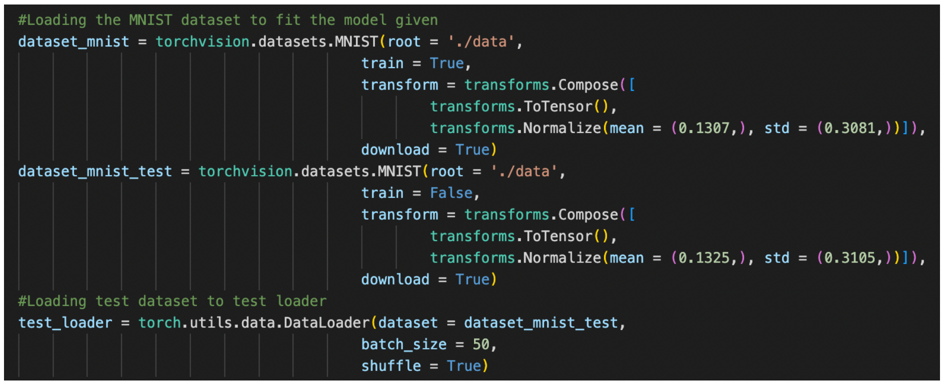
Here I have defined a main function so that execution starts there.
The number of processes is defined and the model is initialized here.
I have used CrossEntropyLoss and SGD optimizer with learning rate of 0.01.



```
#Execution starts here
if __name__== '__main__':
    #Initializing the number of processes
    num_process = 5
    processes = []

    #Initializing the model and sharing the model
    model = Net()
    model.share_memory()

    #Setting the loss function
    cost = nn.CrossEntropyLoss()

    #Setting the optimiser
    optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

The model is shared using model.share_memory()

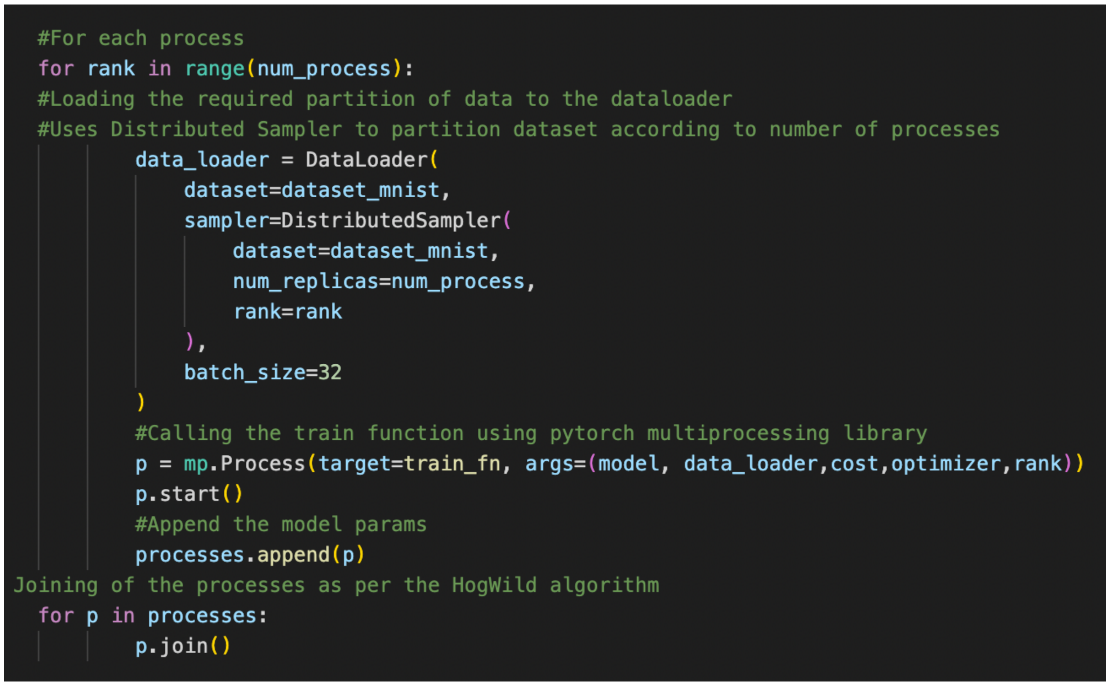
The MNIST dataset is downloaded with data normalisation for train and test set. The testloader loads the test data also.



```
#Loading the MNIST dataset to fit the model given
dataset_mnist = torchvision.datasets.MNIST(root = './data',
                                           train = True,
                                           transform = transforms.Compose([
                                                    transforms.ToTensor(),
                                                    transforms.Normalize(mean = (0.1307,), std = (0.3081,))]),
                                           download = True)
dataset_mnist_test = torchvision.datasets.MNIST(root = './data',
                                           train = False,
                                           transform = transforms.Compose([
                                                    transforms.ToTensor(),
                                                    transforms.Normalize(mean = (0.1325,), std = (0.3105,))]),
                                           download = True)
#Loading test dataset to test loader
test_loader = torch.utils.data.DataLoader(dataset = dataset_mnist_test,
                                           batch_size = 50,
                                           shuffle = True)
```

For each process in the total number of processes defined, we run the following:

- We use DistributedSampler to find the partitions of the dataset for each process. The required partition is then loaded to the Dataloader for training.
- Then we use mp.Process() to call the training function
- All the processes are appended
- Once the training is done for all processes, then join() function is used to find the final parameters.

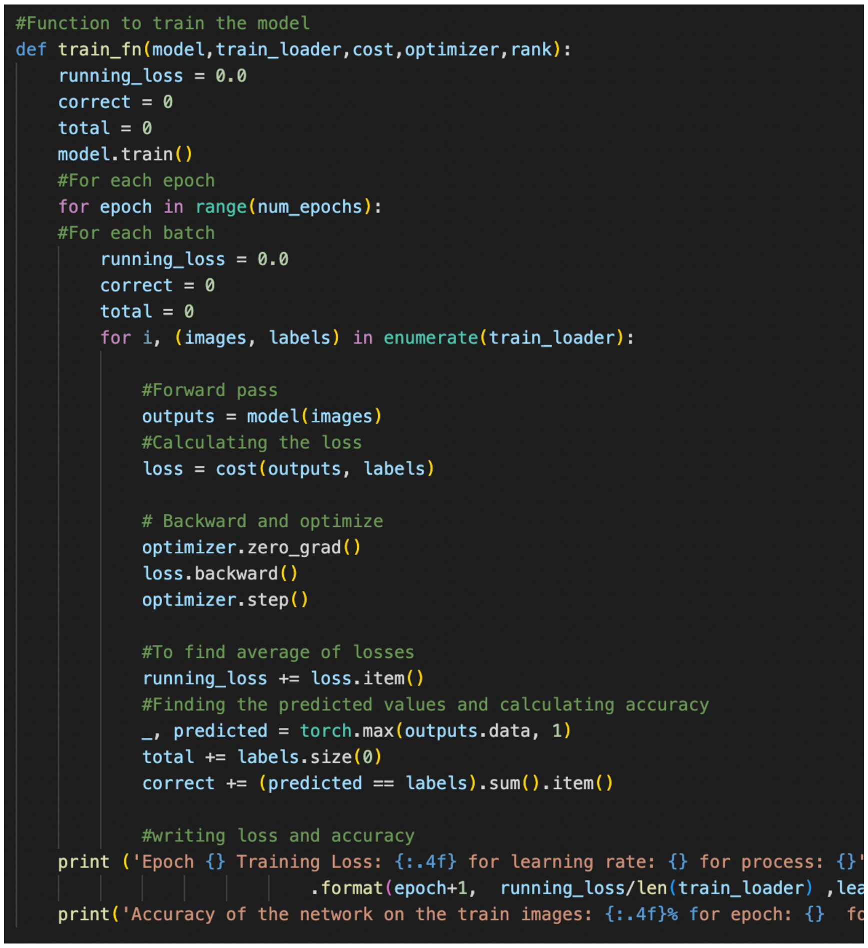


```
  #For each process
  for rank in range(num_process):
  #Loading the required partition of data to the dataloader
  #Uses Distributed Sampler to partition dataset according to number of processes
          data_loader = DataLoader(
              dataset=dataset_mnist,
              sampler=DistributedSampler(
                  dataset=dataset_mnist,
                  num_replicas=num_process,
                  rank=rank
              ),
              batch_size=32
          )
          #Calling the train function using pytorch multiprocessing library
          p = mp.Process(target=train_fn, args=(model, data_loader,cost,optimizer,rank))
          p.start()
          #Append the model params
          processes.append(p)
Joining of the processes as per the HogWild algorithm
  for p in processes:
          p.join()
```

Training Function:

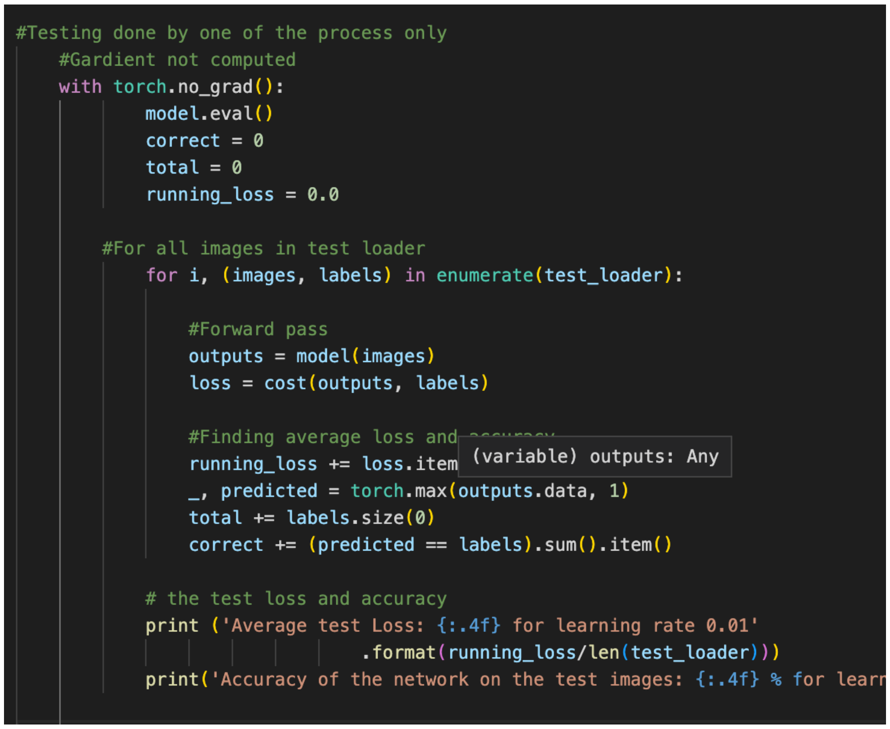
- Training is done similar to the previous question. Model is set to train mode.
- For each epoch, training is carried out
- Iterating through each batch of images
- The outputs are found and loss is propagated backwards

- The loss is appended and average of loss.The accuracy is found by comparing the predicted values and true values.
- The losses and accuracy are printed at the end. Here the convergence criteria taken is the maximum number of iterations - 10.



```
#Function to train the model
def train_fn(model,train_loader,cost,optimizer,rank):
    running_loss = 0.0
    correct = 0
    total = 0
    model.train()
    #For each epoch
    for epoch in range(num_epochs):
    #For each batch
        running_loss = 0.0
        correct = 0
        total = 0
        for i, (images, labels) in enumerate(train_loader):

            #Forward pass
            outputs = model(images)
            #Calculating the loss
            loss = cost(outputs, labels)

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            #To find average of losses
            running_loss += loss.item()
            #Finding the predicted values and calculating accuracy
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            #writing loss and accuracy
    print ('Epoch {} Training Loss: {:.4f} for learning rate: {} for process: {}'
                        .format(epoch+1,  running_loss/len(train_loader) ,lea
    print('Accuracy of the network on the train images: {:.4f}% for epoch: {}  fo
```

Testing is done after the training. We set the model to eval mode.For testing gradient is not computed and losses are also not propagated back. The losses for each image batch is appended and mean is taken to get the test loss. The accuracy of the test set is also found out. The testing is done by one of the processes only. Then the end time is recorded.

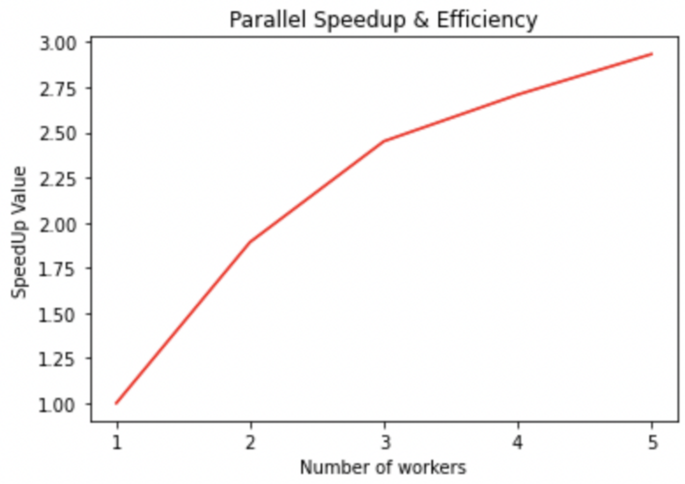


```
#Testing done by one of the process only
    #Gardient not computed
    with torch.no_grad():
            model.eval()
            correct = 0
            total = 0
            running_loss = 0.0

        #For all images in test loader
            for i, (images, labels) in enumerate(test_loader):

                #Forward pass
                outputs = model(images)
                loss = cost(outputs, labels)

                #Finding average loss and
                running_loss += loss.item (variable) outputs: Any
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

            # the test loss and accuracy
            print ('Average test Loss: {:.4f} for learning rate 0.01'
                                .format(running_loss/len(test_loader)))
            print('Accuracy of the network on the test images: {:.4f} % for learn
```

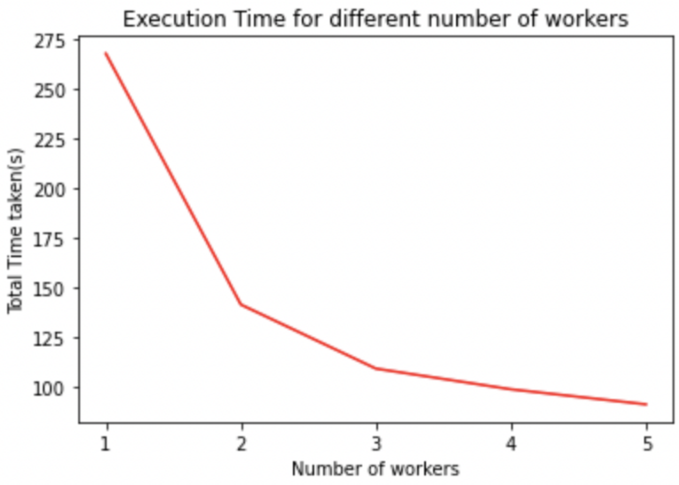
The time taken for running the program for the number of processes, P = 1,2,3,4,5 are recorded as below

Number of workers	Time in seconds(s)
1	268.0132
2	141.6145
3	109.3527
4	98.9295
5	91.3572

SpeedUp graph



Time taken graph

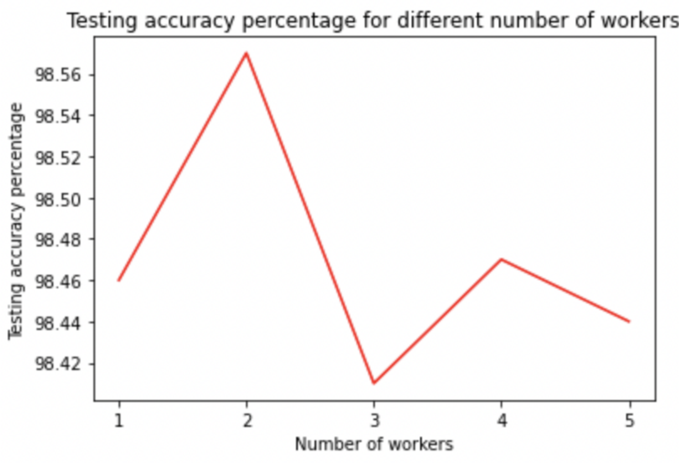


Observation:

We observe that the time decreases as the number of workers increase. There is initially a steep decrease and then it is more gradual. But overall it shows a good speedup.

The time taken by HogWild is also lesser than that taken in the previous exercise. Hence HogWild performs better in terms of time than mpi.

We also plot the testing accuracy of the model for P=1,2,3,4,5 as below

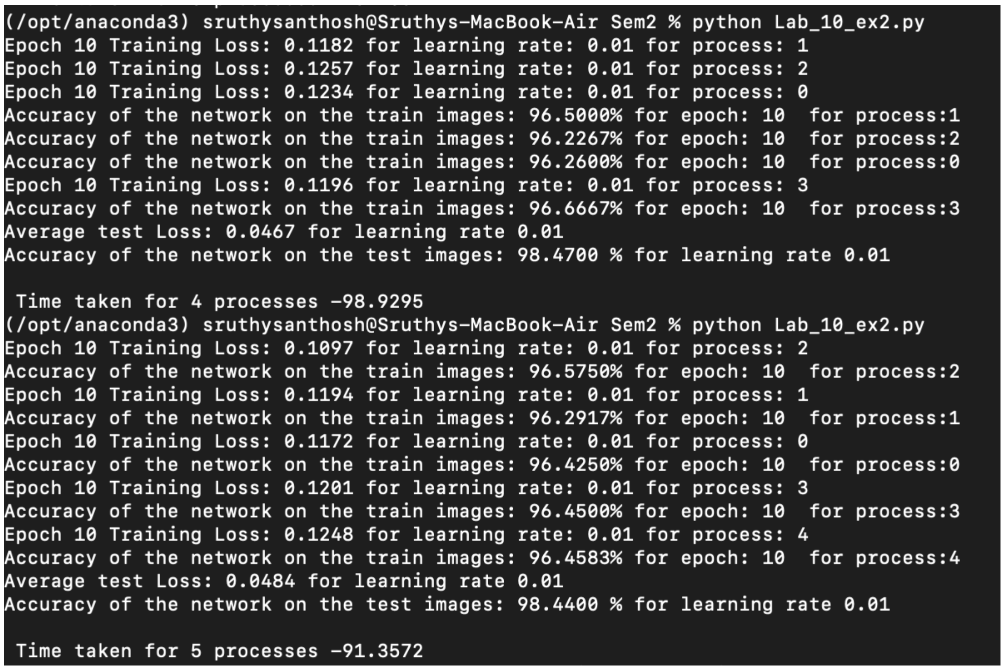


Observation:

Here we see that the accuracy increases, decreases and then again increases as we increase the number of processes. The difference in values is also very less. Hence we can say that there is no significant impact on the accuracy if we increase the number of processes. Hence accuracy is not adversely affected.

We can also see that the accuracy obtained here is better than that obtained in first question. Hence HogWild performs better PSGD in better time also.

Output Screenshot:



```
(/opt/anaconda3) sruthysanthosh@Sruthys-MacBook-Air Sem2 % python Lab_10_ex2.py
Epoch 10 Training Loss: 0.1182 for learning rate: 0.01 for process: 1
Epoch 10 Training Loss: 0.1257 for learning rate: 0.01 for process: 2
Epoch 10 Training Loss: 0.1234 for learning rate: 0.01 for process: 0
Accuracy of the network on the train images: 96.5000% for epoch: 10  for process:1
Accuracy of the network on the train images: 96.2267% for epoch: 10  for process:2
Accuracy of the network on the train images: 96.2600% for epoch: 10  for process:0
Epoch 10 Training Loss: 0.1196 for learning rate: 0.01 for process: 3
Accuracy of the network on the train images: 96.6667% for epoch: 10  for process:3
Average test Loss: 0.0467 for learning rate 0.01
Accuracy of the network on the test images: 98.4700 % for learning rate 0.01

 Time taken for 4 processes -98.9295
(/opt/anaconda3) sruthysanthosh@Sruthys-MacBook-Air Sem2 % python Lab_10_ex2.py
Epoch 10 Training Loss: 0.1097 for learning rate: 0.01 for process: 2
Accuracy of the network on the train images: 96.5750% for epoch: 10  for process:2
Epoch 10 Training Loss: 0.1194 for learning rate: 0.01 for process: 1
Accuracy of the network on the train images: 96.2917% for epoch: 10  for process:1
Epoch 10 Training Loss: 0.1172 for learning rate: 0.01 for process: 0
Accuracy of the network on the train images: 96.4250% for epoch: 10  for process:0
Epoch 10 Training Loss: 0.1201 for learning rate: 0.01 for process: 3
Accuracy of the network on the train images: 96.4500% for epoch: 10  for process:3
Epoch 10 Training Loss: 0.1248 for learning rate: 0.01 for process: 4
Accuracy of the network on the train images: 96.4583% for epoch: 10  for process:4
Average test Loss: 0.0484 for learning rate 0.01
Accuracy of the network on the test images: 98.4400 % for learning rate 0.01

 Time taken for 5 processes -91.3572
```

References:

- https://github.com/xhzhao/PyTorch-MPI-DDP-example/blob/master/mnist_dist.py
- https://yuqli.github.io/jekyll/update/2018/12/17/sgd-mpi-pytorch.html
- https://github.com/yuqli/hpc/blob/master/lab4/src/lab4_multiplenode.py
- https://www.uni-hildesheim.de/learnweb2022/pluginfile.php/82076/mod_resource/content/1/bd-08-sgd.pdf
- https://pytorch.org/tutorials/intermediate/dist_tuto.html
- https://stackoverflow.com/questions/47483385/no-module-named-torch-utils-data-distributed
- https://github.com/pytorch/examples/blob/main/mnist_hogwild/main.py
- https://github.com/wenig/hogwild/blob/master/index.py
- https://towardsdatascience.com/this-is-hogwild-7cc80cd9b944