Lab Course: Distributed Data Analytics
Exercise Sheet 9

Syed Wasif Murtaza Jafri- 311226

Exercise 1: Implementing Parallel Stochastic Gradient Descent

For this lab, you will implement the network following architecture and use the MNIST dataset.

1. conv1: convolution
2. pool1: max pooling
3. rectified linear activation (RELU)
4. conv2: convolution
5. pool2: max pooling
6. rectified linear activation (RELU)
7. dropout
8. pool3: max pooling
9. FC1: fully connected layer
10. rectified linear activation (RELU)
11. FC2: fully connected layer
12. softmax_layer: final output predictions i.e. classify into one of the ten classes.

First having looked at the design of the network and then its regularization, we are now ready to bring our knowledge full circle by going back to where we started in this semester, MPI. PSGD makes use of the simplicity of SGD along with the work division strategies we have learned this semester to arrive at a parallel algorithm that is aimed at speeding up the training of our model. For a detailed description of SGD please refer to the lecture from Lars which can be found on learn web. For this lab, we will be implementing PSGD using pytorch. 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. Steps involved in PSGD can be found on slide 6. Pytorch allows us to access the model weights using the following command "model.named_parameters()".

This command will provide an iterator which can be used to access each weight and bias of the network as shown below:

```
for name, param in model.named_parameters():
    print(name, param)
```

Using the MPI framework, you are required to implement the algorithm on slide 6 of the SGD lecture (from learnweb). Line 3 of the algorithm should be paid special attention to since it is the clue for how the different workers will behave. Each worker needs to calculate the weights until either the termination condition of the max iteration (you can choose how you want to design the training algorithm). Please report the timings of the vanilla SGD ($P = 1$) and for $P = 2, 3, 4, 5$. Show with the help of a speedup graph how adding workers impacts the time needed for PSGD. Please also report on the accuracy of the model for ($P = 1$) and for $P = 2, 3, 4, 5$. Comment on the behavior you see.

First initalizing the model and training and test methods,then with DistributedSampler dividing data among different workers and running train methon on each worker.Each worker return model params to master and then master calculates average of these param and update model param and evaluate model on test data.

In [ ]:

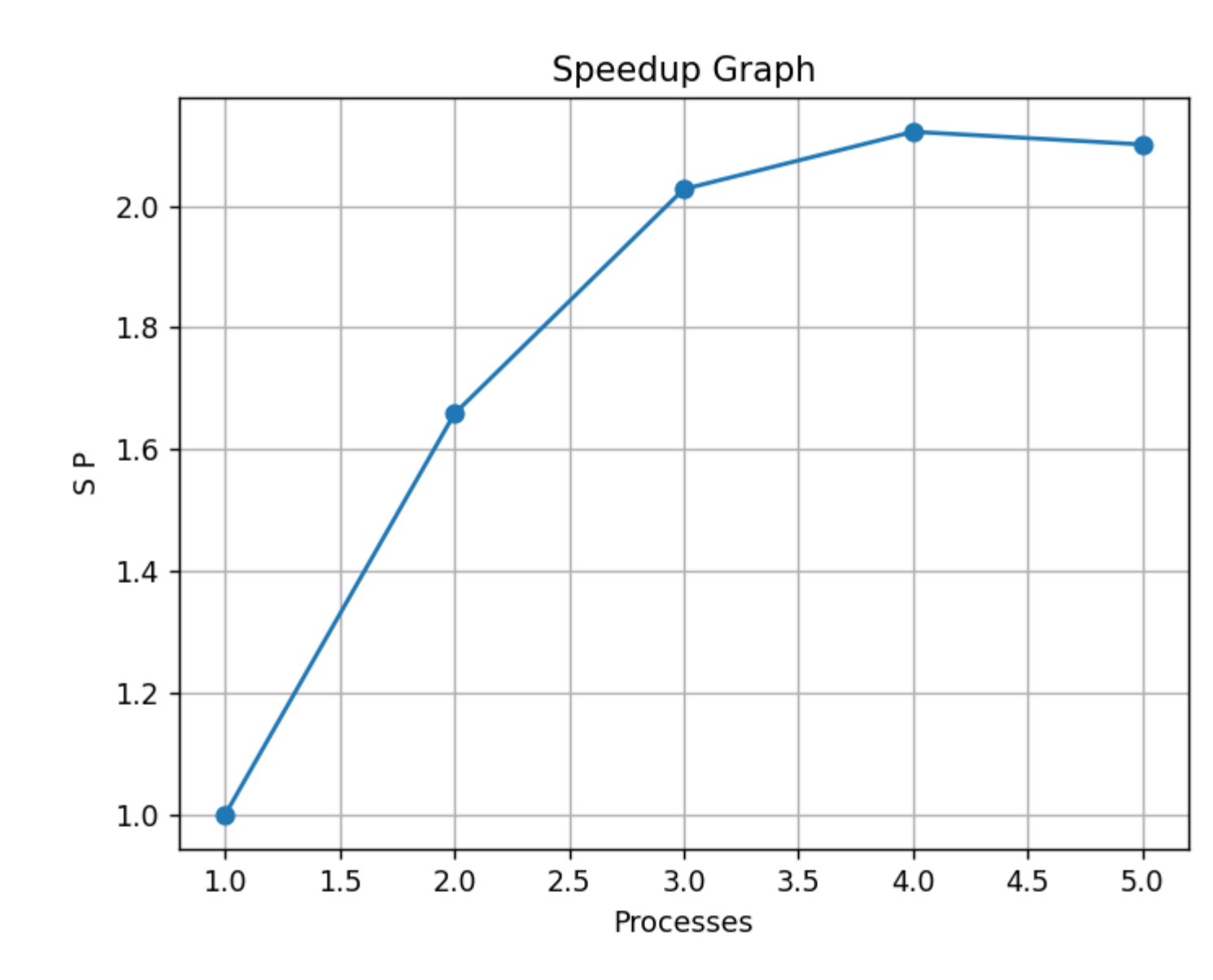
```python
## Exercise1.py
from nltk.corpus import stopwords
from mpi4py import MPI
import warnings
warnings.filterwarnings('ignore')
import torch
import torch.nn as nn
import torchvision.datasets as datasets  # Standard datasets
import torchvision.transforms as transforms  # Transformations we can perform on our dataset for
augmentation
from torch import dropout, optim  # For optimizers like SGD, Adam, etc.
from torch import nn  # All neural network modules
from torch.utils.data import DataLoader,Dataset  # Gives easier dataset managment by creating
mini batches etc.
from torch.utils.tensorboard import SummaryWriter
from torchvision import transforms
import matplotlib.pyplot as plt
torch.cuda.empty_cache()
import torch.nn.functional as F
import numpy as np
from timeit import default_timer as timer

comm = MPI.COMM_WORLD # setting up the MPI communicator
total_worker = comm.Get_size() # getting number of workers
rank = comm.Get_rank() # storing rank of each worker
start_time = MPI.Wtime()  # time variable to find time at the end
master = 0  # master with id=0

class CNN(nn.Module):
    def __init__(self, input_channels=1, num_of_classes=10,dropout=0.5):
        super(CNN, self).__init__()
        self.relu = nn.ReLU()
        self.softmax = nn.LogSoftmax()
        self.dropout = dropout
        self.conv1 = nn.Conv2d(
            in_channels = input_channels,
            out_channels = 16,
            kernel_size = (5,5),
            stride=(1,1),
            padding=(0,0)

        )
        self.pool1 = nn.MaxPool2d(kernel_size=(2,2),stride=(2,2))
        self.conv2 = nn.Conv2d(
            in_channels = 16,
            out_channels = 32,
            kernel_size = (5,5),
            stride=(1,1),
            padding=(0,0)
        )
        self.pool2 = nn.MaxPool2d(kernel_size=(2,2),stride=(2,2))
        self.dropout1 = nn.Dropout(self.dropout)
        self.pool3 = nn.MaxPool2d(kernel_size=(2,2),stride=(2,2))
        self.FC1 = nn.Linear(32* 2* 2,512)
        self.FC2 = nn.Linear(512,256)
        self.Output = nn.Linear(256,num_of_classes)


    def forward(self,x):

        x = self.conv1(x)
        x = self.relu(self.pool1(x))
        x = self.conv2(x)
        x = self.relu(self.pool2(x))
        x = self.dropout1(x)
        x = self.pool3(x)
        x = x.view(-1, 32* 2* 2)

        x = self.relu(self.FC1(x))
        x = self.FC2(x)

        x = self.softmax(self.Output(x))

        return x
def accuracy(outputs, labels):
    ''' Calculating accuray of prediction'''

    _, preds = torch.max(outputs, dim = 1) # class with highest probablity is the predicted
output
    return(torch.tensor(torch.sum(preds == labels).item()/ len(preds)))

def trainModel(model,optimizer,criterion,train_loader,num_epochs):

    for epoch in range(num_epochs):
        lossSum = 0
        accSum = 0

        # iterating through all batches
        for i,sample_batched in enumerate(train_loader):

            # setting gradient to zero before forward pass
            optimizer.zero_grad()

            # model predict
            yhat=model(sample_batched[0])

            # model loss
            loss=criterion(yhat.squeeze(),sample_batched[1].squeeze())
            lossSum += loss.item()

            # model accuracy
            acc = accuracy(yhat.squeeze(),sample_batched[1].squeeze())

            accSum += acc.item()

            # calculating gradient through backprop
            loss.backward(retain_graph=True)

            # updating parameters
            optimizer.step()
                    # writing accuracy and loss to tensorboard logs
        lossAvg = lossSum /len(train_loader)
        accAvg = accSum/len(train_loader)



        if (epoch+1) % 10 == 0:
            print (f'Epoch [{epoch+1}/{num_epochs}],Train Loss: {lossAvg:.4f},Train Accuracy:
{accAvg:.4f}')

    print('Finished Training at rank',rank)
    return model
def evalModel(model,criterion,test_loader):
    accSum = 0
    lossSum = 0
    # for evaluating model on test data with no gradient calculation
    with torch.no_grad():
        for i,sample_batched in enumerate(test_loader):

            # model predict
            yhat=model(sample_batched[0])

            # model loss
            loss=criterion(yhat.squeeze(),sample_batched[1].squeeze())
            lossSum += loss
            # model acc
            acc = accuracy(yhat.squeeze(),sample_batched[1].squeeze())
            accSum += acc

            # writing accuracy and loss to tensorboard logs

    totalloss = lossSum/len(test_loader)
    totalAccuracy = accSum/len(test_loader)
    print('\nTest Data: Average loss: {:.4f}, Final Accuracy: ({:.0f}%)\n'.format(totalloss,100.
* totalAccuracy))


    return totalAccuracy.item(),totalloss.item()



# setting device
# device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device = 'cpu'
torch.device(device)

in_channels = 1 # input channels of image
num_classes = 10 # number of output class [0-10]
num_epochs = 50
batch_size= 256

transform = transforms.Compose([transforms.Resize((32,32)),
    transforms.ToTensor()
])

# getting training and test data
train_dataset = datasets.MNIST(root="MNIST/", train=True, transform=transform)
test_dataset = datasets.MNIST(root="MNIST/", train=False, transform=transform)
train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset,
num_replicas=total_worker, rank=rank)

train_loader = DataLoader(dataset=train_dataset,sampler=train_sampler,batch_size=batch_size)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size ,shuffle=False)

# different setting of learning rates
learning_rate =0.1

criterion = nn.NLLLoss()

# Initialize network
model = CNN(input_channels=in_channels, num_of_classes=num_classes,dropout=0.4).to(device)


# Initializing Optimizer
optimizer= torch.optim.SGD(model.parameters(),lr=learning_rate)
print('-----------------------------------------------------------------')
print('|--------------Experiment: Number of processes='+str(total_worker)+' ---------------|')
print('-----------------------------------------------------------------')

model = trainModel(model, optimizer,criterion, train_loader, num_epochs)

# for workers, creating a dictionary of parameters and sending to master
if rank != 0:
    param_dict = {}
    for name,param in model.named_parameters():
        param_dict.update({name:param.data.detach()})

    comm.send(param_dict,0)

if rank == 0:
    # for master creating a dictionary of parameters of model of master
    param_dict = {}
    for name,param in model.named_parameters():
        param_dict.update({name:param.data.detach()})

    # recieving model param dict from workers and adding each layer params with master
    for i in range(1,total_worker):
        model_params = comm.recv(source=i)
        for key in model_params:
            paramValue = torch.add(model_params.get(key),param_dict.get(key))
            param_dict.update({key:paramValue})

    # taking average by dividing  number of workers
    for key in param_dict:
        paramValue = param_dict.get(key) / total_worker
        param_dict.update({key:paramValue})

    print('Updating params....')
    # updating parameters with average paramters of all processes
    with torch.no_grad():
        for name, param in model.named_parameters():
            param.copy_(torch.tensor(param_dict.get(name)))

    # calculating time for training
    total_time = MPI.Wtime() - start_time
    print('Total time for training: {}'.format(total_time))

    # evaluating model on test data
    testAcc,testLoss = evalModel(model,criterion,test_loader)

    # storing time and accuracy for speedup and accuracy graphs
    mode = 'w'  if total_worker==1 else 'a'
    with open('TimeforTraining.txt', mode) as f:
        f.write(str(total_worker) +','+str(total_time)+','+str(testAcc*100) + '\n')
```

```
-----------------------------------------------------------------
|-------------Experiment: Number of processes=1 --------------|
-----------------------------------------------------------------
Epoch [10/50],Train Loss: 0.0739,Train Accuracy: 0.9762
Epoch [20/50],Train Loss: 0.0447,Train Accuracy: 0.9860
Epoch [30/50],Train Loss: 0.0353,Train Accuracy: 0.9887
Epoch [40/50],Train Loss: 0.0298,Train Accuracy: 0.9904
Epoch [50/50],Train Loss: 0.0250,Train Accuracy: 0.9916
Finished Training at rank 0
Updating params....
Total time for training: 915.6394292999757
```

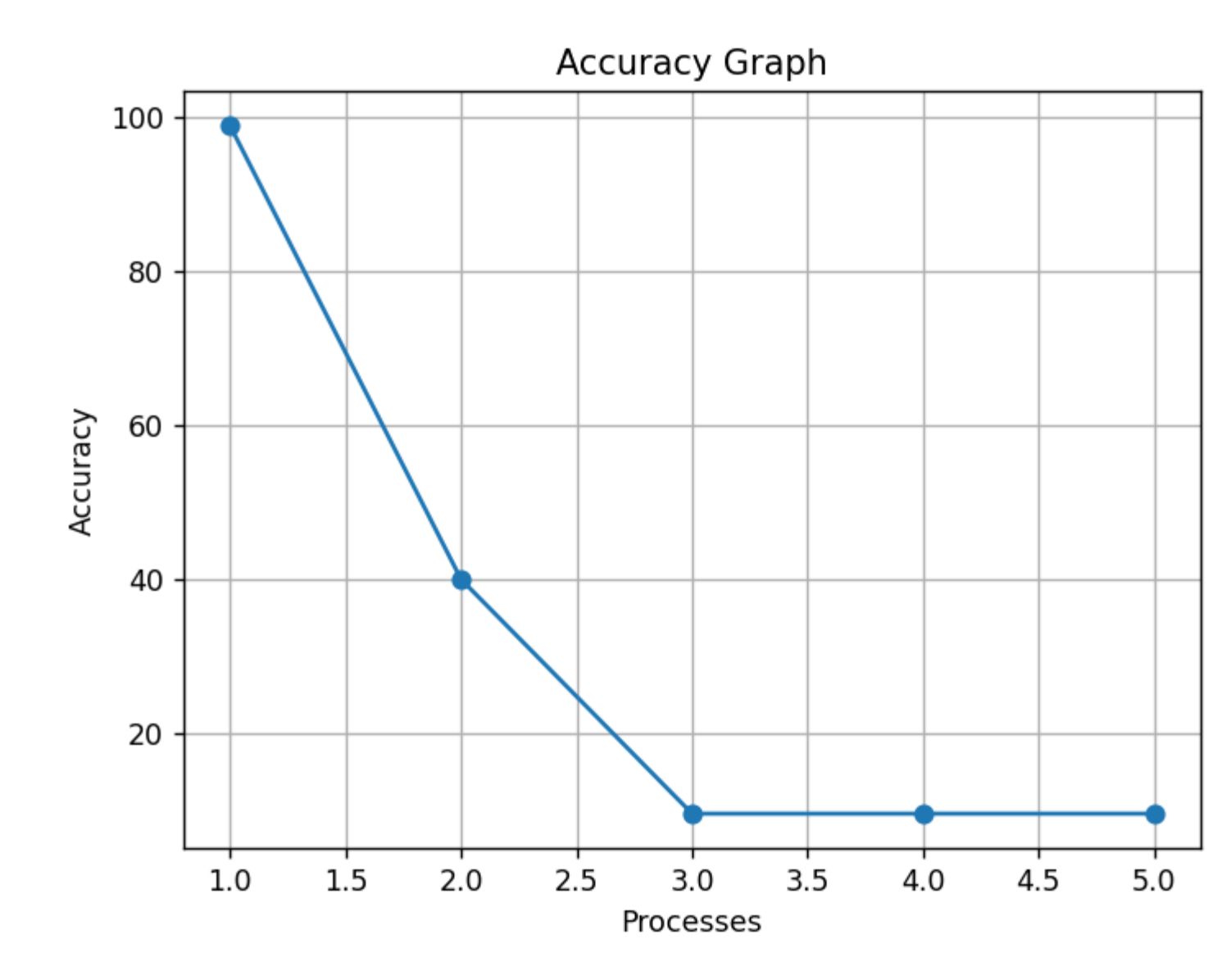
```
-----------------------------------------------------------------
|-------------Experiment: Number of processes=2 --------------|
-----------------------------------------------------------------
Epoch [10/50],Train Loss: 0.0956,Train Accuracy: 0.9690
Epoch [20/50],Train Loss: 0.0611,Train Accuracy: 0.9810
Epoch [30/50],Train Loss: 0.0476,Train Accuracy: 0.9845
Epoch [40/50],Train Loss: 0.0370,Train Accuracy: 0.9881
Epoch [50/50],Train Loss: 0.0291,Train Accuracy: 0.9907
Finished Training at rank 1
-----------------------------------------------------------------
|-------------Experiment: Number of processes=2 --------------|
-----------------------------------------------------------------
Epoch [10/50],Train Loss: 0.1069,Train Accuracy: 0.9655
Epoch [20/50],Train Loss: 0.0635,Train Accuracy: 0.9794
Epoch [30/50],Train Loss: 0.0492,Train Accuracy: 0.9837
Epoch [40/50],Train Loss: 0.0368,Train Accuracy: 0.9879
Epoch [50/50],Train Loss: 0.0333,Train Accuracy: 0.9883
Finished Training at rank 0
Updating params....
Total time for training: 552.05229589995

Test Data: Average loss: 1.9407, Final Accuracy: (40%)
```

In [ ]:

```python
import numpy as np
from matplotlib import pyplot as plt


speedupList = []

with open('TimeforTraining.txt', 'r') as f:
    Lines = f.readlines()
    timeList = []
    print(Lines)
    for line in Lines:
        timeList.append([float(x) for x in line.strip().split(',')])

workers = np.array(timeList)[:,0]
executionTime = np.array(timeList)[:,1]
accuracy = np.array(timeList)[:,2]

ts = executionTime[0]
sp = []

for i in range(len(workers)):
    sp.append(ts/executionTime[i])


plt.plot(workers, sp,marker='o')
plt.title('Speedup Graph')
plt.xlabel('Processes')
plt.ylabel('S P')
plt.grid()
plt.show()

plt.plot(workers, accuracy,marker='o')
plt.title('Accuracy Graph')
plt.xlabel('Processes')
plt.ylabel('Accuracy')
plt.grid()
plt.show()
```



As we increase the number of processes, our speedup increases rapidly in beginning than decreases after inceasing over 4 workers.



With the increase in number of workers,accuracy at each workers is good, but after taking average test accuracy is decreasing with more workers.

PyTorch distributed execution

For this exercise, we will now let PyTorch handle the multiprocessing nature of the execution for us. You are required to implement HogWild SGD but this time using the torch.multiprocessing library. We will be using a share memory architecture here for a comprehensive explanation of shared memory models/archs and multiprocessing in PyTorch please refer to the annex. The algorithm for HogWild can be found on slide 12 of the lecture (on learnweb). Please note that by letting PyTorch handle the multiprocessing and using a shared-memory architecture, you do not have to manually implement lines 7 to 9. Similar to the previous exercise, please report your speedup graphs and your final accuracies for the models and comment on any discrepancies you notice.

In this exercise,we are impelementing pytorch Multiprocessing library.Starting each worker process by passing distributed dataset and train function.Then merging with processes with join.All the processes have shared memory to update model paramters.

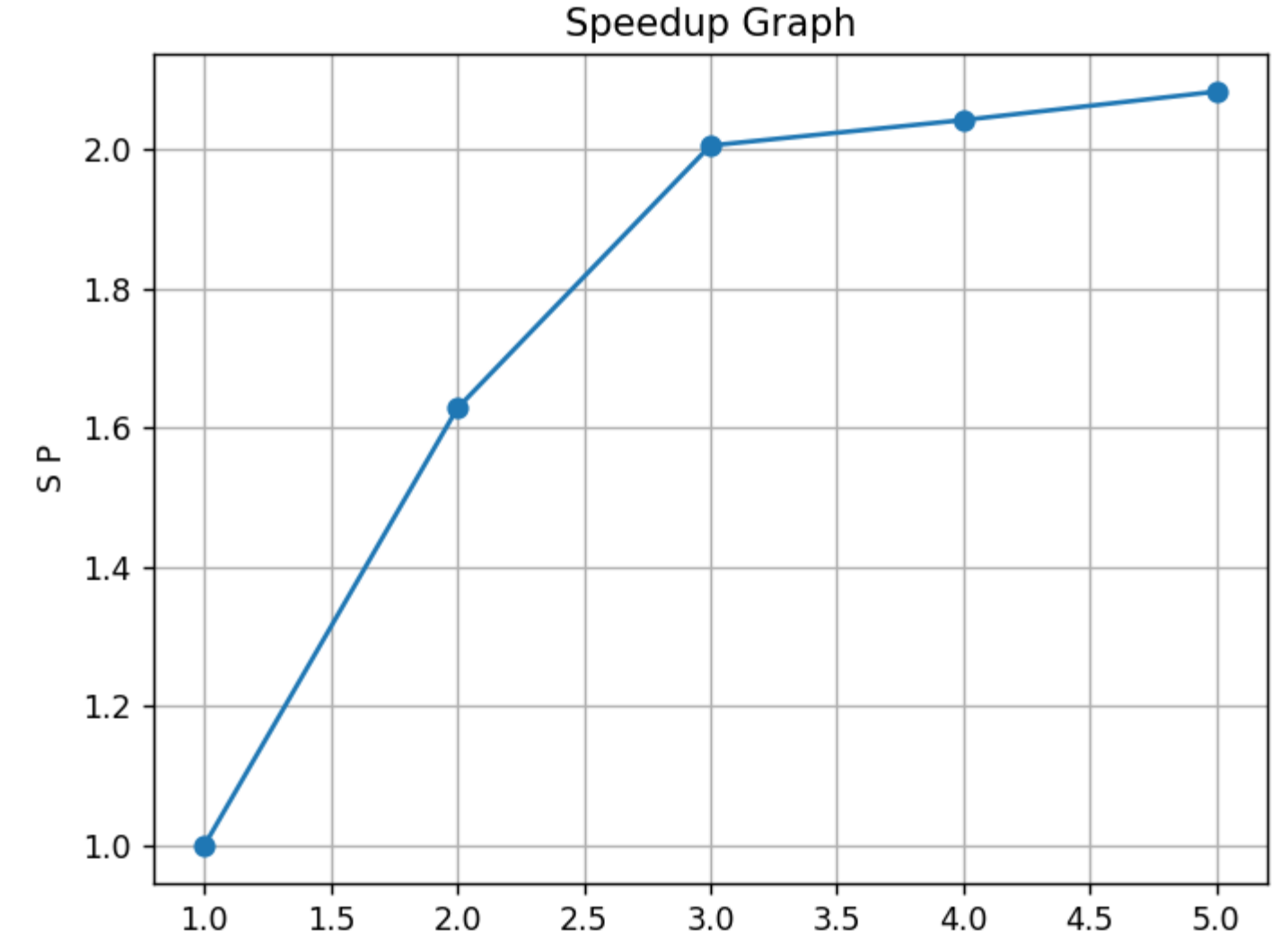
In [ ]:

```python
## Exercise2.py

from nltk.corpus import stopwords
from mpi4py import MPI
import warnings
warnings.filterwarnings('ignore')
import torch
import torch.nn as nn
import torchvision.datasets as datasets  # Standard datasets
import torchvision.transforms as transforms  # Transformations we can perform on our dataset for augmentation
from torch import dropout, optim  # For optimizers like SGD, Adam, etc.
from torch import nn  # All neural network modules
from torch.utils.data import DataLoader,Dataset  # Gives easier dataset managment by creating mini batches etc.
from torch.utils.tensorboard import SummaryWriter
from torchvision import transforms
import matplotlib.pyplot as plt
torch.cuda.empty_cache()
import torch.nn.functional as F
import numpy as np
from timeit import default_timer as timer
import torch.multiprocessing as mp
import time
import sys


class CNN(nn.Module):
    def __init__(self, input_channels=1, num_of_classes=10,dropout=0.5):
        super(CNN, self).__init__()
        self.relu = nn.ReLU()
        self.softmax = nn.LogSoftmax()
        self.dropout = dropout
        self.conv1 = nn.Conv2d(
            in_channels = input_channels,
            out_channels = 16,
            kernel_size = (5,5),
            stride=(1,1),
            padding=(0,0)

        )
        self.pool1 = nn.MaxPool2d(kernel_size=(2,2),stride=(2,2))
        self.conv2 = nn.Conv2d(
            in_channels = 16,
            out_channels = 32,
            kernel_size = (5,5),
            stride=(1,1),
            padding=(0,0)
        )
        self.pool2 = nn.MaxPool2d(kernel_size=(2,2),stride=(2,2))
        self.dropout1 = nn.Dropout(self.dropout)
        self.pool3 = nn.MaxPool2d(kernel_size=(2,2),stride=(2,2))
        self.FC1 = nn.Linear(32* 2* 2,512)
        self.FC2 = nn.Linear(512,256)
        self.Output = nn.Linear(256,num_of_classes)


    def forward(self,x):

        x = self.conv1(x)
        x = self.relu(self.pool1(x))
        x = self.conv2(x)
        x = self.relu(self.pool2(x))
        x = self.dropout1(x)
        x = self.pool3(x)
        x = x.view(-1, 32* 2* 2)

        x = self.relu(self.FC1(x))
        x = self.FC2(x)

        x = self.softmax(self.Output(x))

        return x
def accuracy(outputs, labels):
    ''' Calculating accuray of prediction'''

    _, preds = torch.max(outputs, dim = 1) # class with highest probablity is the predicted output
    return(torch.tensor(torch.sum(preds == labels).item()/ len(preds)))

def trainModel(model,learning_rate,criterion,train_loader,num_epochs,rank):
    # Initializing Optimizer
    optimizer= torch.optim.SGD(model.parameters(),lr=learning_rate)

    for epoch in range(num_epochs):
        lossSum = 0
        accSum = 0

        # iterating through all batches
        for i,sample_batched in enumerate(train_loader):

            # setting gradient to zero before forward pass
            optimizer.zero_grad()

            # model predict
            yhat=model(sample_batched[0])

            # model loss
            loss=criterion(yhat.squeeze(),sample_batched[1].squeeze())
            lossSum += loss.item()

            # model accuracy
            acc = accuracy(yhat.squeeze(),sample_batched[1].squeeze())
            accSum += acc.item()

            # calculating gradient through backprop
            loss.backward(retain_graph=True)

            # updating parameters
            optimizer.step()
                    # writing accuracy and loss to tensorboard logs
        lossAvg = lossSum /len(train_loader)
        accAvg = accSum/len(train_loader)


    print (f'Worker: [{rank}],Train Loss: {lossAvg:.4f},Train Accuracy: {accAvg:.4f}')

    print('Finished Training at rank:',rank)
    return model
def evalModel(model,criterion,test_loader):
    accSum = 0
    lossSum = 0

    # for evaluating model on test data with no gradient calculation
    with torch.no_grad():
        for i,sample_batched in enumerate(test_loader):

            # model predict
            yhat=model(sample_batched[0])

            # model loss
            loss=criterion(yhat.squeeze(),sample_batched[1].squeeze())
            lossSum += loss
            # model acc
            acc = accuracy(yhat.squeeze(),sample_batched[1].squeeze())
            accSum += acc

            # writing accuracy and loss to tensorboard logs

    totalloss = lossSum/len(test_loader)
    totalAccuracy = accSum/len(test_loader)
    print('\nTest Data: Average loss: {:.4f}, Final Accuracy: ({:.0f}%)\n'.format(totalloss,100.
* totalAccuracy))


    return totalAccuracy.item(),totalloss.item()


if __name__ == '__main__':
    # setting device
    device =  "cpu"
    torch.device(device)

    in_channels = 1 # input channels of image
    num_classes = 10 # number of output class [0-10]
    num_epochs = 30
    batch_size= 256
    transform = transforms.Compose([transforms.Resize((32,32)),
    transforms.ToTensor()
    ])

    # getting training and test data
    train_dataset = datasets.MNIST(root="MNIST/", train=True, transform=transform)
    test_dataset = datasets.MNIST(root="MNIST/", train=False, transform=transform)
    test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size ,shuffle=False)

    # different setting of learning rates
    learning_rate =0.001

    criterion = nn.NLLLoss()

    # Initialize network
    model = CNN(input_channels=in_channels, num_of_classes=num_classes,dropout=0.4).to(device)



    # initialize model
    mp.set_start_method('spawn')
    model = CNN(input_channels=in_channels, num_of_classes=num_classes,dropout=0.4).to(device)
    model.share_memory()
    num_processes = 5
    processes = []
    start_time = time.perf_counter()
    print('----------------------------------------------------------------')
    print('|--------------Experiment: Number of processes='+str(num_processes)+' --------------
|')
    print('----------------------------------------------------------------')
    for rank in range(num_processes):
        train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset,
num_replicas=num_processes, rank=rank)
        train_loader =
DataLoader(dataset=train_dataset,sampler=train_sampler,batch_size=batch_size)
        ## initializing and starting p processes
        p = mp.Process(target=trainModel, args=(model, learning_rate,criterion, train_loader,
num_epochs,rank))
        p.start()
        processes.append(p)
    for p in processes:
        p.join()
    total_time = time.perf_counter() - start_time

    # calculating time for training
    total_time = MPI.Wtime() - start_time
    print('Total time for training: {}'.format(total_time))


    # evaluating model on test data
    testAcc,testLoss = evalModel(model,criterion,test_loader)
    # storing time and accuracy for speedup and accuracy graphs
    mode = 'w'  if num_processes==1 else 'a'
    with open('TimeforTraining2.txt', mode) as f:
        f.write(str(num_processes) +','+str(total_time)+','+str(testAcc*100) + '\n')
```

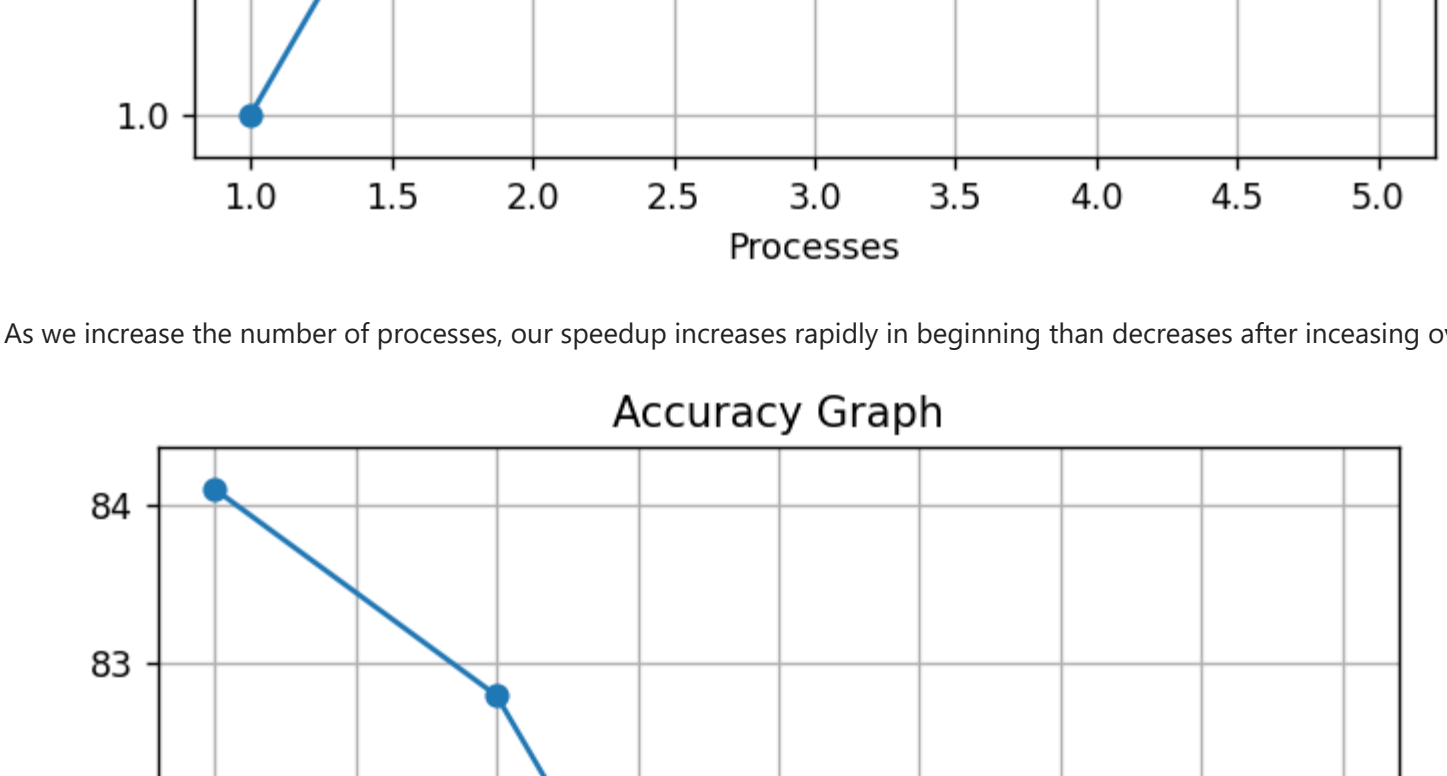
```
PS C:\Users\wasif\DDA Lab\lab09> & "C:/Program Files/Python310/python.exe" "c:/Users/wasif/DDA Lab/lab09/Exercise2.py"
----------------------------------------------------------------
|--------------Experiment: Number of processes=2 --------------|
----------------------------------------------------------------
Worker: [1],Train Loss: 0.5980,Train Accuracy: 0.8166
Finished Training at rank: 1
Worker: [0],Train Loss: 0.6012,Train Accuracy: 0.8169
Finished Training at rank: 0
Total time for training: 338.57162439997774

Test Data: Average loss: 0.5532, Final Accuracy: (83%)
```

In [ ]:

```python
## Exercise2Analysis.py

import numpy as np
from matplotlib import pyplot as plt


speedupList = []

with open('TimeforTraining2.txt', 'r') as f:
    Lines = f.readlines()
    timeList = []
    print(Lines)
    for line in Lines:
        timeList.append([float(x) for x in line.strip().split(',')])

print(np.array( timeList))
workers = np.array(timeList)[:,0]
executionTime = np.array(timeList)[:,1]
accuracy = np.array(timeList)[:,2]

ts = executionTime[0]
sp = []

for i in range(len(workers)):
    sp.append(ts/executionTime[i])


plt.plot(workers, sp,marker='o')
plt.title('Speedup Graph')
plt.xlabel('Processes')
plt.ylabel('S P')
plt.grid()
plt.show()

plt.plot(workers, accuracy,marker='o')
plt.title('Accuracy Graph')
plt.xlabel('Processes')
plt.ylabel('Accuracy')
plt.grid()
plt.show()
```



As we increase the number of processes, our speedup increases rapidly in beginning than decreases after inceasing over 4 workers.



Accuracy does not change much with increasing number of workers as compared to MPI version.