CS 547: Deep Learning, Fall 2019

Homework 4: Residual Neural Network

Professor: Justin Sirignano

1. Goal

You will learn how to build very deep convolutional networks using Residual Networks (ResNets). In theory, deep networks can represent very complex functions. However, in practice, they are difficult to train. Residual Networks, introduced by He et al. (2015), allow for training deeper networks than was previously possible. In this homework assignment, you have the following task:

1. You will implement ResNets from scratch and test it on CIFAR100 and Tiny ImageNet dataset via Pytorch.

2. You will implement ResNets with *distributed* stochastic gradient descent on 2-4 nodes via Pytorch in Blue Waters.

   - Synchronous stochastic gradient descent
   - *Asynchronous stochastic gradient descent (10 bonus points)

3. You will also learn how to use a pre-trained ResNet (which was trained on the ImageNet dataset) and then fine-tune it on CIFAR100. This is an example of *transfer learning*.

2. Dataset

We provide the CIFAR100 and Tiny ImageNet datasets. They are located in the following directory:

`/projects/training/bayw/HW4_Dataset/`

2.1 CIFAR100

1. Copy the CIFAR100 dataset folder into your own directory

```
cp -r /projects/training/bayw/HW4_Dataset/cifar-100-python ~/scratch/.
```

2. Load CIFAR100 dataset

   - Use the following code to create dataloader

```
# For trainning data
trainset = torchvision.datasets.CIFAR100(root='~/scratch/',
        train=True,download=False, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset,
        batch_size=batch_size, shuffle=True, num_workers=8)
# For testing data
testset = torchvision.datasets.CIFAR100(root='~/scratch/',
        train=False,download=False, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset,
        batch_size=batch_size, shuffle=False, num_workers=8)
```

   - Once you have created dataloader, you can access the data with the following code

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
for epoch in range(num_epochs):
    for batch_idx, (images, labels) in enumerate(trainloader):
        images = images.to(device)
        labels = labels.to(device)
```

2.2 TinyImageNet

1. Copy the TinyImageNet dataset folder into your own directory

```
cp /projects/training/bayw/HW4_Dataset/tiny-imagenet-200.zip ~/scratch/.
```

   Note: Please copy the dataset to your own directory. DO NOT use the data in shared folder directly or unzip it in the shared directory to reduce the IO burden.

2. Unzip the file with the following command. (It may take a while to unzip it.)

```
unzip ~/scratch/tiny-imagenet-200.zip
```

3. Use the following code to create dataloader

```
def create_val_folder(val_dir):
    """
    This method is responsible for separating validation
    images into separate sub folders
    """
    # path where validation data is present now
    path = os.path.join(val_dir, 'images')
    # file where image2class mapping is present
    filename = os.path.join(val_dir, 'val_annotations.txt')
    fp = open(filename, "r")  # open file in read mode
    data = fp.readlines()  # read line by line

    '''
    Create a dictionary with image names as key and
    corresponding classes as values
    '''
    val_img_dict = {}
    for line in data:
        words = line.split("\t")
        val_img_dict[words[0]] = words[1]
    fp.close()
    # Create folder if not present, and move image into proper folder
    for img, folder in val_img_dict.items():
        newpath = (os.path.join(path, folder))
        if not os.path.exists(newpath):  # check if folder exists
            os.makedirs(newpath)
        # Check if image exists in default directory
        if os.path.exists(os.path.join(path, img)):
            os.rename(os.path.join(path, img), os.path.join(newpath, img))
    return
```

```
train_dir = '/u/training/YOUR_BW_ID/scratch/tiny-imagenet-200/train'
train_dataset = datasets.ImageFolder(train_dir,
         transform=transform_train)
#print(train_dataset.class_to_idx)
train_loader = torch.utils.data.DataLoader(train_dataset,
        batch_size=batch_size, shuffle=True, num_workers=8)

val_dir = '/u/training/YOUR_BW_ID/scratch/tiny-imagenet-200/val/images'
if 'val_' in os.listdir(val_dir)[0]:
    create_val_folder(val_dir)
else:
    pass

val_dataset = datasets.ImageFolder(val_dir,
        transform=transforms.ToTensor())
#print(val_dataset.class_to_idx)
val_loader = torch.utils.data.DataLoader(val_dataset,
        batch_size=batch_size, shuffle=False, num_workers=8)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
for images, labels in train_loader:
    images = Variable(images).to(device)
    labels = Variable(labels).to(device)

for images, labels in val_loader:
    images = Variable(images).to(device)
    labels = Variable(labels).to(device)
```

Note: There are many ways to create a dataloader for TinyImageNet dataset, and I use *torchvision.datasets.ImageFolder*. Basically, *ImageFolder* load the images in different directories. You can consider the following example. There are two classes, i.e., cat and dog. And under the root directory, there should be two directories. One is named dog and the other is named cat. All images of dog are in the *dog* directory and all the images of cat are in the *cat* directory. Moreover, *ImageFolder* use *os.walk* function to iterate all the files including the files in the sub-directory under the folder of each class. For example, it will include the file $dog4.png$ which is located in $root/dog/sub-directory/dog4.png$. Furthermore, *ImageFolder* uses *classes.sort*() for class assignment, which means if the names of each class folder is fixed, the assignment is fixed.

```
root/dog/dog1.png
root/dog/dog2.png
```

```
root/dog/dog3.png
root/cat/cat1.png
root/cat/cat2.png
root/cat/cat3.png
```

And you can check the assignment of each class by

```
print(train_dataset.class_to_idx)
```

3. Implement ResNets from Scratch

1. Implement basic block described in Fig 1 with the following code layout

```
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        # YOUR CODE
    def forward(self, x):
       # YOUR CODE
```

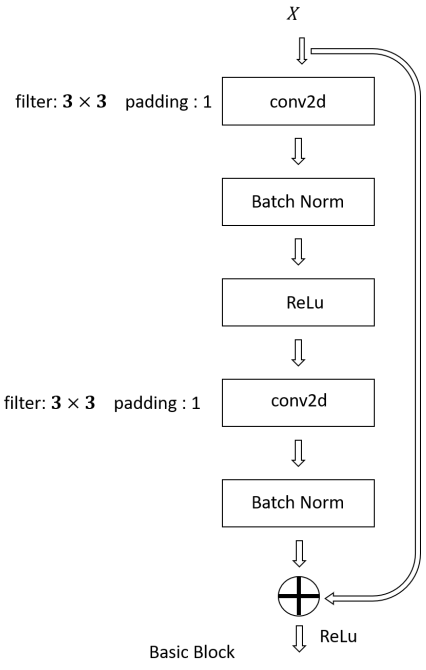

Figure 1: Basic Block

2. Build the ResNets described in Fig 2 with your basic block.

```
class ResNet(nn.Module):
    def __init__(self, basic_block, num_blocks, num_classes):
        super(ResNet, self).__init__()
        # YOUR CODE

    def forward(self, x):
    # YOUR CODE
```

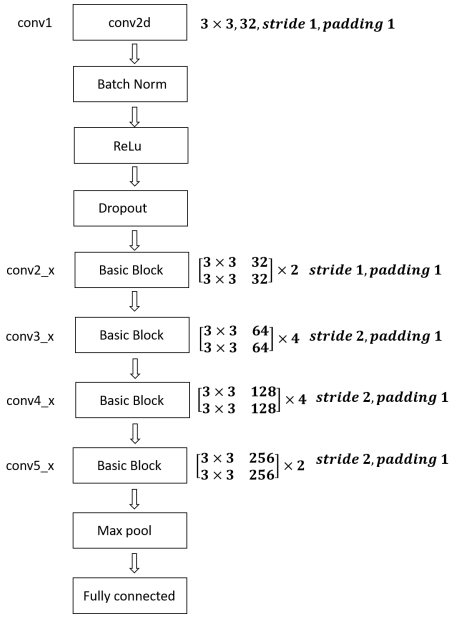

Figure 2: ResNet Structure

3. Train and test your model on CIFAR100 dataset and TinyImageNet dataset as usual. We recommend you to plot learning curve of trainning and testing accuracy.

Note: You should follow the hyperparameters mentioned in the Figures, otherwise you will get points off.

4. Train ResNets with Distributed SGD

In order to do distributed training, we need MPI to support it. Indeed, Pytorch suppports three different backends(GLOO,MPI and NCCL). On Blue Waters, we use MPI and MPI for python is supported in python version ***bwpy*** different version from your current one ***python/2.0.1***. However, in ***bwpy***, the Pytorch version is 0.3.0 and some functions supported in Pytorch 0.4.0 are not available in Pytorch 0.3.0. For example, in Pytorch 0.3.0, you should use ***.cuda()*** instead of ***.to('cuda')*** and ***device()*** is not supported in Pytorch 0.3.0. Moreover,***python-mpi*** are incompatible with ***multiprocessing***. Thus if you want to use the Pytorch ***dataloader***, you should force one of the arguments ***num_worker*** to be 0.

First, we will go over how to write bash script for multiple nodes and then give your sample code on how to do initialization on Pytorch distributed training. Second, we will go over some requirement for your distributed training part.

Note: You may choose to use 2 to 4 nodes for the distributed training. The more nodes you choose, the longer waiting time for queuing it could be.

4.1 How to setup

4.1.1 How to setup the bash script

```
#!/bin/bash
#PBS -l nodes=02:ppn=16:xk
#PBS -l walltime=06:00:00
#PBS -N sync_sgd_cifar100
#PBS -e $PBS_JOBID.err
#PBS -o $PBS_JOBID.out
#PBS -m bea
#PBS -M YOUR_EMAIL
cd #YOUR CODE DIRECTORY
. /opt/modules/default/init/bash # NEEDED to add module commands to shell
module load bwpy
module load bwpy-mpi
aprun -n 2 -N 1 python sync_sgd_cifar100.py
```

Note that the following line is to declare how many nodes you need for your job. If you need 10 nodes, then change 2 to 10 accordingly.

```
#PBS -l nodes= 2:ppn=16:xk
```

```
aprun -n 2 -N 1 python sync_sgd_cifar100.py
```

4.1.1 How to do initialization for pytorch distributed training on Blue Waters

There are different ways to do initialization on distributed training in Pytorch for different situations. If you are not using Blue Waters for distributed training, you can refer the following links https://pytorch.org/docs/stable/distributed.html.

If are using Blue Waters, you can just use the following code.

```
import torch.distributed as dist
import os
import subprocess
from mpi4py import MPI
cmd = "/sbin/ifconfig"
out, err = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE,
    stderr=subprocess.PIPE).communicate()
ip = str(out).split("inet addr:")[1].split()[0]
name = MPI.Get_processor_name()
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
num_nodes = int(comm.Get_size())
ip = comm.gather(ip)
if rank != 0:
  ip = None
ip = comm.bcast(ip, root=0)
os.environ['MASTER_ADDR'] = ip[0]
os.environ['MASTER_PORT'] = '2222'
backend = 'mpi'
dist.init_process_group(backend, rank=rank, world_size=num_nodes)
dtype = torch.FloatTensor
```

4.2 Distributed Training

4.2.1 Synchronous SGD

Data: CIFAR100
Result: Classification Model on CIFAR100
Do Synchronous Initialization for All servers;
for *Server = 0,1,2,...K-1 (Do in Parallel)* **do**
  for *epoch = 0,1,2,...N-1* **do**
    for *each mini-batch data* **do**
      - Calculate gradient for each server
      - AllReduce to synchronous the gradient of all servers.
      - Update weight of the model.

    end
  end
end

Algorithm 1: Synchronous SGD

The graph below is a demonstration of AllReduce Operation, and following function is for the AllReduce operation from Pytorch Distributed.

```
dist.all_reduce()
```

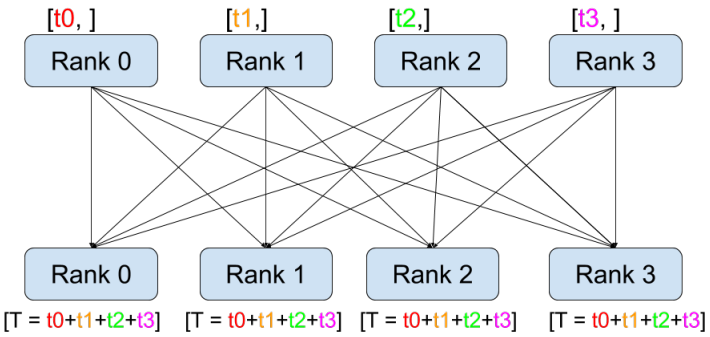

Figure 3: AllReduce Operation

*4.2.2 Asynchronous SGD

Note: This part is for bonus point

Data: CIFAR100
Result: Classification Model on CIFAR100
Do Synchronous Initialization for All servers;
Server 0 to be parameter server;
Server 1...K-1 to be worker servers;
for *Server = 1,2,...K-1 (Do in Parallel)* **do**
  for *epoch = 0,1,2,...N-1* **do**
    for *each mini-batch data* **do**
      - Calculate gradient for each server
      - Send the gradient to parameter server.
      - Get the updated parameter from parameter server and update worker's own parameter.

    end
  end
end
Note:

1. For parameter server, it should receive gradients from each worker server whenever they send the request.
2. Use ***dist.isend*** and ***dist.irecv*** to send and receive tensors.

Algorithm 2: Asynchronous SGD

5. Fine-tune a Pretrained ResNets

1. Copy the pre-trained model file into your code directory with the following command

```
cp -r /projects/training/bayw/HW4_Model/ ~/HW4/.
```

2. Use the followng code to load the pre-trained ResNets18.

```
def resnet18(pretrained=True):
    model = torchvision.models.resnet.ResNet(
            torchvision.models.resnet.BasicBlock, [2, 2, 2, 2])
    if pretrained:
        model.load_state_dict(model_zoo.load_url(
                model_urls['resnet18'],model_dir='./'))
```

```
    return model
model = resnet18(pretrained=True)
```

3. Since input size for the pre-trained model is 224×224, and you have to up-sample the images in CIFAR100 dataset with the help of the following function from *torch.nn*.

```
nn.Upsample(scale_factor=7, mode='bilinear')
```

   Moreover, the pre-trained model was trained on ImageNet dataset, and you should change the output layer such that the dimension matches the dimension of CIFAR100 dataset.

4. Train and test your model on CIFAR100 dataset as usual. We recommend you to plot learning curve of trainning and testing accuracy.

6. Homework Requirements

6.1 Performance Requirements

In order to get a full credit, you need to achieve at least the accuracy mentioned below for each task except for ResNet CIFAR100 with asynchronous SGD. ResNet CIFAR100 with asynchronous SGD is a 10-point bonus question.

	Accuracy	Expected Running time on Blue Waters
ResNet CIFAR100	60%	2 hour
ResNet TinyImageNet	50%	4 hours
Pre-trained ResNet CIFAR100	70%	1 hour
ResNet CIFAR100 with synchronous SGD	60%	2 hour
*ResNet CIFAR100 with asynchronous SGD	60%	2 hours

6.2 What to submit

- For each task you should submit an individual main file.
  1. `resnet_cifar100.py`
  2. `resnet_tinyimagenet.py`
  3. `pretrained_cifar100.py`
  4. `sync_sgd_cifar100.py`

5. `async_sgd_cifar100.py`

Note: You can have extra python files including some helper functions, but please make sure you have an individual main file for each tasks.

- For each task, you should report your achieved test accuracy and plot the learning curve. The x-axis of your plot is epoch numbers and y-axis of your plot is test accuracy.
- We provide partial code in the following directory, which basically covers the code mentioned above. It is not required to use these code, but you can use it as a reference.

```
/projects/training/bayw/HW4
```

7. References

1. Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun - Deep Residual Learning for Image Recognition (2015)
2. Pytorch torchvision.models https://pytorch.org/docs/stable/torchvision/models.html
3. Pytorch distributed Training https://pytorch.org/docs/stable/distributed.html