Deep Learning

AR

Human Action Recognition

This assignment will work with the UCF-101 human action recognition dataset. The original technical report published in 2012 discussing the dataset can be found here. The dataset consists of 13,320 videos between ~2-10 seconds long of humans performing one of 101 possible actions. The dimensions of each frame are 320 by 240.

CNNs have been shown to work well in nearly all tasks related to images. A video can be thought of simply as a sequence of images. That is, a network designed for learning from videos must be able to handle the spatial information as well as the temporal information (typically referred to as learning spatiotemporal features). There are many modeling choices to deal with this. Below are a list of papers mentioned in the action recognition presentation given in class. This list is in no way comprehensive but definitely gives a good idea of the general progress.

Large-scale Video Classification with Convolutional Neural Networks (2014)

Two-Stream Convolutional Networks for Action Recognition in Videos (2014)

Beyond Short Snippets: Deep Networks for Video Classification (2015)

Learning Spatiotemporal Features with 3D Convolutional Networks (2015)

The Kinetics Human Action Video Dataset (2017)

Quo Vadis, Action Recognition? A New Model and the Kinetics Dataset (2017)

Can Spatiotemporal 3D CNNs Retrace the History of 2D CNNs and ImageNet? (2017)

This homework will compare a single frame model (spatial information only) with a 3D convolution based model (2 spatial dimensions + 1 temporal dimension).

- Part one: Fine-tune a 50-layer ResNet model (pretrained on ImageNet) on single UCF-101 video frames
- Part two: Fine-tune a 50-layer 3D ResNet model (pretrained on Kinetics) on UCF-101 video sequences

The original dataset (which can be downloaded here) is relatively small (9.5GB). However, it can be extremely slow to load and decode video files from memory on BlueWaters. Because of this, all of the videos have been stored as numpy arrays of size `[sequence_length,height,width,3]`. This makes them very quick to load but extremely large. The directory `/projects/training/bayw/hdf5/UCF-101-hdf5/` contains all of these files and it is ~500GB.

This code uses the more recent version of PyTorch meaning the command `module load bwpy/2.0.0-pre2` must be used on BlueWaters.

helperFunctions.py

Create a new directory and a python script `helperFunctions.py`. In here, you will place the a few functions which are related to information about the dataset and data loading. These functions will be used in place of a PyTorch dataset/dataloader. Include the following import statements.

```
import os
import numpy as np
```

```
import cv2
import time
import h5py
```

getUCF101

```
def getUCF101(base_directory = ''):

    # action class labels
    class_file = open(base_directory + 'ucfTrainTestlist/classInd.txt','r')
    lines = class_file.readlines()
    lines = [line.split(' ')[1].strip() for line in lines]
    class_file.close()
    class_list = np.asarray(lines)

    # training data
    train_file = open(base_directory + 'ucfTrainTestlist/trainlist01.txt','r')
    lines = train_file.readlines()
    filenames = ['UCF-101/' + line.split(' ')[0] for line in lines]
    y_train = [int(line.split(' ')[1].strip())-1 for line in lines]
    y_train = np.asarray(y_train)
    filenames = [base_directory + filename for filename in filenames]
    train_file.close()

    train = (np.asarray(filenames),y_train)

    # testing data
    test_file = open(base_directory + 'ucfTrainTestlist/testlist01.txt','r')
    lines = test_file.readlines()
    filenames = ['UCF-101/' + line.split(' ')[0].strip() for line in lines]
    classnames = [filename.split('/')[1] for filename in filenames]
    y_test = [np.where(classname == class_list)[0][0] for classname in classnames]
    y_test = np.asarray(y_test)
    filenames = [base_directory + filename for filename in filenames]
    test_file.close()

    test = (np.asarray(filenames),y_test)

    return class_list, train, test
```

There is a text file provided with the dataset which lists all of the relative paths for the videos for the train/test split. This function can be called with the location of the dataset (`/projects/training/bayw/hdf5/UCF-101-hdf5/`) and returns three variables. `class_list` is a list of the action categories. `train` is a tuple. The first element is a numpy array with the absolute filepaths for the videos for training. The second element is a numpy array of class indices (0-100). `test` is a tuple in the same format as `train` but for the test dataset.

loadFrame

```
def loadFrame(args):
    mean = np.asarray([0.485, 0.456, 0.406],np.float32)
    std = np.asarray([0.229, 0.224, 0.225],np.float32)

    curr_w = 320
    curr_h = 240
    height = width = 224
    (filename,augment) = args

    data = np.zeros((3,height,width),dtype=np.float32)

    try:
        ### load file from HDF5
        filename = filename.replace('.avi','.hdf5')
        filename = filename.replace('UCF-101','UCF-101-hdf5')
```

```
            h = h5py.File(filename,'r')
            nFrames = len(h['video']) - 1
            frame_index = np.random.randint(nFrames)
            frame = h['video'][frame_index]

            if(augment==True):
                ## RANDOM CROP - crop 70-100% of original size
                ## don't maintain aspect ratio
                if(np.random.randint(2)==0):
                    resize_factor_w = 0.3*np.random.rand()+0.7
                    resize_factor_h = 0.3*np.random.rand()+0.7
                    w1 = int(curr_w*resize_factor_w)
                    h1 = int(curr_h*resize_factor_h)
                    w = np.random.randint(curr_w-w1)
                    h = np.random.randint(curr_h-h1)
                    frame = frame[h:(h+h1),w:(w+w1)]

                ## FLIP
                if(np.random.randint(2)==0):
                    frame = cv2.flip(frame,1)

                frame = cv2.resize(frame,(width,height))
                frame = frame.astype(np.float32)

                ## Brightness +/- 15
                brightness = 30
                random_add = np.random.randint(brightness+1) - brightness/2.0
                frame += random_add
                frame[frame>255] = 255.0
                frame[frame<0] = 0.0

            else:
                # don't augment
                frame = cv2.resize(frame,(width,height))
                frame = frame.astype(np.float32)

            ## resnet model was trained on images with mean subtracted
            frame = frame/255.0
            frame = (frame - mean)/std
            frame = frame.transpose(2,0,1)
            data[:,:,:] = frame
        except:
            print("Exception: " + filename)
            data = np.array([])
        return data
```

The above function is used for loading a single frame from a particular video in the dataset. `args` is a tuple with the first argument being the location of a video (which is in `train[0]` and `test[0]` from the `getUCF101()` function) and the second argument specifies whether data augmentation should be performed. The video paths in `train` and `test` look something like this: `/projects/training/bayw/hdf5/UCF-101-hdf5/FloorGymnastics/v_FloorGymnastics_g23_c01.avi`. However, since the dataset has now been saved as hdf5 files, the first part of this code converts this to `/projects/training/bayw/hdf5/UCF-101/FloorGymnastics/v_FloorGymnastics_g23_c01.hdf5`. A random frame is selected and if `augment==True`, the frame is randomly cropped, resized to the appropriate dimension for the model, flipped, and has its brightness adjusted. The frame is normalized based on the provided `mean` and `std` for the default PyTorch pretrained ResNet-50 model. Finally the data is returned and is a numpy array of size `[3,224,224]` of type `np.float32`. This is used for part 1.

loadSequence

```
def loadSequence(args):
    mean = np.asarray([0.433, 0.4045, 0.3776],np.float32)
    std = np.asarray([0.1519876, 0.14855877, 0.156976],np.float32)
```

```
        curr_w = 320
        curr_h = 240
        height = width = 224
        num_of_frames = 16

        (filename,augment) = args

        data = np.zeros((3,num_of_frames,height,width),dtype=np.float32)

        try:
            ### load file from HDF5
            filename = filename.replace('.avi','.hdf5')
            filename = filename.replace('UCF-101','UCF-101-hdf5')
            h = h5py.File(filename,'r')
            nFrames = len(h['video']) - 1
            frame_index = np.random.randint(nFrames - num_of_frames)
            video = h['video'][frame_index:(frame_index + num_of_frames)]

            if(augment==True):
                ## RANDOM CROP - crop 70-100% of original size
                ## don't maintain aspect ratio
                resize_factor_w = 0.3*np.random.rand()+0.7
                resize_factor_h = 0.3*np.random.rand()+0.7
                w1 = int(curr_w*resize_factor_w)
                h1 = int(curr_h*resize_factor_h)
                w = np.random.randint(curr_w-w1)
                h = np.random.randint(curr_h-h1)
                random_crop = np.random.randint(2)

                ## Random Flip
                random_flip = np.random.randint(2)

                ## Brightness +/- 15
                brightness = 30
                random_add = np.random.randint(brightness+1) - brightness/2.0

                data = []
                for frame in video:
                    if(random_crop):
                        frame = frame[h:(h+h1),w:(w+w1),:]
                    if(random_flip):
                        frame = cv2.flip(frame,1)
                    frame = cv2.resize(frame,(width,height))
                    frame = frame.astype(np.float32)

                    frame += random_add
                    frame[frame>255] = 255.0
                    frame[frame<0] = 0.0

                    frame = frame/255.0
                    frame = (frame - mean)/std
                    data.append(frame)
                data = np.asarray(data)

            else:
                # don't augment
                data = []
                for frame in video:
                    frame = cv2.resize(frame,(width,height))
                    frame = frame.astype(np.float32)
                    frame = frame/255.0
                    frame = (frame - mean)/std
                    data.append(frame)
                data = np.asarray(data)

            data = data.transpose(3,0,1,2)
        except:
            print("Exception: " + filename)
            data = np.array([])
        return data
```

This is very similar to the `loadFrame()` function. The 3D ResNet-50 model used for part 2 is trained on sequences of length 16. This function simply grabs a random subsequence of frames and augments them all in the exact same way (this is important when performing data augmentation on videos). This function returns a numpy array of size `[3,16,224,224]`. The last three channels must be the time and space dimensions since the PyTorch 3D convolution implementation acts on the last three channels of an input with size `[batch_size,num_of_input_features,time,height,width]`.

Part 1 - Single Frame Model

This first portion of the homework uses a pretrained ResNet-50 model pretrained on ImageNet. Although the dataset has a large number of frames (13000 videos * number of frames per video), the frames are correlated with eachother meaning there isn't a whole lot of variety. Also, to keep the sequences relatively short (~2-10 seconds), some of the original videos were split up into 5-6 shorter videos meaning there is even less variety. Single frames alone can still provide a significant amount of information about the action being performed (consider the classes "Skiing" versus "Baseball Pitch"). Training a CNN from scratch significantly overfits. However, the features from a CNN pretrained on ImageNet (over 1 million images of 1000 classes) can be very useful even in video based problem like action recognition. A single frame model performs surprisingly well. This doesn't necessarily mean solving the task of learning from images inherently solves all video related tasks. It's more likely that with the problem of human action recognition, the spatial information is more important than the temporal information.

```
import numpy as np
import os
import sys
import time

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
#from torchvision import datasets, transforms
from torch.autograd import Variable
import torch.distributed as dist
import torchvision

from helperFunctions import getUCF101
from helperFunctions import loadFrame

import h5py
import cv2

from multiprocessing import Pool

IMAGE_SIZE = 224
NUM_CLASSES = 101
batch_size = 100
lr = 0.0001
num_of_epochs = 10


data_directory = '/projects/training/bayw/hdf5/UCF-101-hdf5/'
class_list, train, test = getUCF101(base_directory = data_directory)
```

Import the necessary python modules, set a few basic hyperparameters, and load the dataset variables `class_list`, `train`, and `test`.

```
model =  torchvision.models.resnet50(pretrained=True)
model.fc = nn.Linear(2048,NUM_CLASSES)
```

```
for param in model.parameters():
    param.requires_grad_(False)

# for param in model.conv1.parameters():
#     param.requires_grad_(True)
# for param in model.bn1.parameters():
#     param.requires_grad_(True)
# for param in model.layer1.parameters():
#     param.requires_grad_(True)
# for param in model.layer2.parameters():
#     param.requires_grad_(True)
# for param in model.layer3.parameters():
#     param.requires_grad_(True)
for param in model.layer4[2].parameters():
    param.requires_grad_(True)
for param in model.fc.parameters():
    param.requires_grad_(True)

params = []
# for param in model.conv1.parameters():
#     params.append(param)
# for param in model.bn1.parameters():
#     params.append(param)
# for param in model.layer1.parameters():
#     params.append(param)
# for param in model.layer2.parameters():
#     params.append(param)
# for param in model.layer3.parameters():
#     params.append(param)
for param in model.layer4[2].parameters():
    params.append(param)
for param in model.fc.parameters():
    params.append(param)

model.cuda()

optimizer = optim.Adam(params,lr=lr)
criterion = nn.CrossEntropyLoss()
```

The module `torchvision` comes with a pretrained ResNet-50 model. Overwrite the last fully connected layer such that it has the number of outputs equal to the number of classes. As mentioned before, the dataset is not large enough to warrant training a full ResNet-50 model. We will just fine-tune the output layer and the last residual block. `model.layer4` contains three residual blocks implying `model.layer4[2]` is the last of these three residual blocks. Fine-tuning only the top layers also reduces the amount of GPU memory meaning a higher batch size can be used and the model can be trained more quickly with less overfitting.

```
pool_threads = Pool(8,maxtasksperchild=200)
```

As mentioned previously, this code does not use a PyTorch dataset/dataloader. However, we can still leverage multiple CPU cores using a `Pool()` object with our dataloading function `loadFrame()`. How this is done will become apparent in the training loop.

```
for epoch in range(0,num_of_epochs):

    ###### TRAIN
    train_accu = []
    model.train()
    random_indices = np.random.permutation(len(train[0]))
    start_time = time.time()
    for i in range(0, len(train[0])-batch_size,batch_size):

        augment = True
        video_list = [(train[0][k],augment)
```

```
                        for k in random_indices[i:(batch_size+i)]]
            data = pool_threads.map(loadFrame,video_list)

            next_batch = 0
            for video in data:
                if video.size==0: # there was an exception, skip this
                    next_batch = 1
            if(next_batch==1):
                continue

            x = np.asarray(data,dtype=np.float32)
            x = Variable(torch.FloatTensor(x)).cuda().contiguous()

            y = train[1][random_indices[i:(batch_size+i)]]
            y = torch.from_numpy(y).cuda()

            output = model(x)

            loss = criterion(output, y)
            optimizer.zero_grad()

            loss.backward()
            optimizer.step()

            prediction = output.data.max(1)[1]
            accuracy = ( float( prediction.eq(y.data).sum() ) /float(batch_size))*100.0
            if(epoch==0):
                print(i,accuracy)
            train_accu.append(accuracy)
        accuracy_epoch = np.mean(train_accu)
        print(epoch, accuracy_epoch,time.time()-start_time)

    torch.save(model,'single_frame.model')
    pool_threads.close()
    pool_threads.terminate()
```

There is nothing particularly unusual about this training loop besides the data loading.

```
augment = True
video_list = [(train[0][k],augment)
              for k in random_indices[i:(batch_size+i)]]
data = pool_threads.map(loadFrame,video_list)

next_batch = 0
for video in data:
    if video.size==0: # there was an exception, skip this
        next_batch = 1
if(next_batch==1):
    continue

x = np.asarray(data,dtype=np.float32)
x = Variable(torch.FloatTensor(x)).cuda().contiguous()
```

This portion of the code grabs a batch of data by first creating a list of tuples with the tuples being video filepaths (`train[0][k]`) and data augmentation (`augment=True`). The `pool_threads.map()` function takes as input a function (`loadFrame()`) and a list of arguments (`video_list`) for this function. The function is called for every tuple in the list. The `Pool()` object was created with the ability to run execute these commands on eight separate CPU cores. The final output is going to be a list of data frames (size `[3,224,224]`). Some of the frames from the hdf5 files can be corrupt which causes an exception every so often. The second part of the code is simply a hack way of skipping over a particular iteration if any of the function calls fails. Lastly, the list of data frames is converted to a PyTorch variable of size `[batch_size,3,224,224]` and moved to the GPU so it can be input to the ResNet-50 model.

```
    ##### TEST
    model.eval()
    test_accu = []
    random_indices = np.random.permutation(len(test[0]))
    t1 = time.time()
    for i in range(0,len(test[0])-batch_size,batch_size):
        augment = False
        video_list = [(test[0][k],augment)
                      for k in random_indices[i:(batch_size+i)]]
        data = pool_threads.map(loadFrame,video_list)

        next_batch = 0
        for video in data:
            if video.size==0: # there was an exception, skip this batch
                next_batch = 1
        if(next_batch==1):
            continue

        x = np.asarray(data,dtype=np.float32)
        x = Variable(torch.FloatTensor(x)).cuda().contiguous()

        y = test[1][random_indices[i:(batch_size+i)]]
        y = torch.from_numpy(y).cuda()

        output = model(x)

        prediction = output.data.max(1)[1]
        accuracy = ( float( prediction.eq(y.data).sum() ) /float(batch_size))*100.0
        test_accu.append(accuracy)
        accuracy_test = np.mean(test_accu)
    print('Testing',accuracy_test,time.time()-t1)
```

You can also add a portion of code to test the model every epoch (notice `augment=False`). After ~10 epochs, the model will be achieving almost 100% on the training data. To back up the statement before about the frames within a video being highly correlated, after 10 epochs, this means the model has only seen 10 random frames from each video. However, it can correctly classify any other random unseen frame in the training data with near perfect accuracy while having only around 73%-75% accuracy on the test data.

Testing

This test accuracy is only for a single frame. At true test time, it would make sense to average out the prediction over every frame within a video. After training, let's create a loop to calculate predictions for the entire test dataset. This can be done in a separate file or after the training loop.

```
    model = torch.load('single_frame.model')
    model.cuda()

    ##### save predictions directory
    prediction_directory = 'UCF-101-predictions/'
    if not os.path.exists(prediction_directory):
        os.makedirs(prediction_directory)
    for label in class_list:
        if not os.path.exists(prediction_directory+label+'/'):
            os.makedirs(prediction_directory+label+'/')
```

The above creates a directory structure laid out in the exact same way the dataset is structured. Predictions for each video can be saved as numpy arrays stored in hdf5 format and the `train` and `test` list can be used to load these in at a future time.

```
    acc_top1 = 0.0
    acc_top5 = 0.0
    acc_top10 = 0.0
```

```
confusion_matrix = np.zeros((NUM_CLASSES,NUM_CLASSES),dtype=np.float32)
random_indices = np.random.permutation(len(test[0]))
mean = np.asarray([0.485, 0.456, 0.406],np.float32)
std = np.asarray([0.229, 0.224, 0.225],np.float32)
model.eval()
```

Create some variables to track the `(top 1,top 5,top 10)` accuracy. The `top_10` accuracy simply signifies how often the correct class is any of the top 10 predicted classes. A confusion matrix is a square matrix of size `[NUM_CLASSES,NUM_CLASSES]` . The confusion matrix is used to show how often each particular class is classified by the model as other classes. That is, for each video, the value of `confusion_matrix[actual_class,predicted_class]` is increased by 1. After going through the entire dataset, if you divide each row by its row sum, it will provide a percentage breakdown of this class confusion. A perfect classifier would have 100% for every diagonal element and 0% for all of the off diagonal elements.

```
for i in range(len(test[0])):

    t1 = time.time()

    index = random_indices[i]

    filename = test[0][index]
    filename = filename.replace('.avi','.hdf5')
    filename = filename.replace('UCF-101','UCF-101-hdf5')

    h = h5py.File(filename,'r')
    nFrames = len(h['video'])

    data = np.zeros((nFrames,3,IMAGE_SIZE,IMAGE_SIZE),dtype=np.float32)

    for j in range(nFrames):
        frame = h['video'][j]
        frame = frame.astype(np.float32)
        frame = cv2.resize(frame,(IMAGE_SIZE,IMAGE_SIZE))
        frame = frame/255.0
        frame = (frame - mean)/std
        frame = frame.transpose(2,0,1)
        data[j,:,:,:] = frame
    h.close()
```

Start the loop off by loading in the full video into the numpy array `data` of size `[sequence_length,3,224,224]` .

```
    prediction = np.zeros((nFrames,NUM_CLASSES),dtype=np.float32)

    loop_i = list(range(0,nFrames,200))
    loop_i.append(nFrames)

    for j in range(len(loop_i)-1):
        data_batch = data[loop_i[j]:loop_i[j+1]]

        with torch.no_grad():
            x = np.asarray(data_batch,dtype=np.float32)
            x = Variable(torch.FloatTensor(x)).cuda().contiguous()

            output = model(x)

        prediction[loop_i[j]:loop_i[j+1]] = output.cpu().numpy()
```

Continue the loop with the code above. The numpy array `predictions` will hold the probabilities for each class for each frame. It is possible to loop through the frames one at a time and grab the output from the model. However, this would be very slow and not utilize any batch processing. You could

process the full video at once (with the `sequence_length` basically being the `batch_size`), but some of the sequences are too long to fit in memory. This loop breaks the video into subsequences of length 200 (which does fit on the GPU), performs batch processing, sets the `prediction` variable equal to the output for the corresponding frames and continues until the full video has been passed through the model.

```
    filename = filename.replace(data_directory+'UCF-101-hdf5/',prediction_directory)
    if(not os.path.isfile(filename)):
        with h5py.File(filename,'w') as h:
            h.create_dataset('predictions',data=prediction)

    # softmax
    for j in range(prediction.shape[0]):
        prediction[j] = np.exp(prediction[j])/np.sum(np.exp(prediction[j]))

    prediction = np.sum(np.log(prediction),axis=0)
    argsort_pred = np.argsort(-prediction)[0:10]

    label = test[1][index]
    confusion_matrix[label,argsort_pred[0]] += 1
    if(label==argsort_pred[0]):
        acc_top1 += 1.0
    if(np.any(argsort_pred[0:5]==label)):
        acc_top5 += 1.0
    if(np.any(argsort_pred[:]==label)):
        acc_top10 += 1.0

    print('i:%d nFrames:%d t:%f (%f,%f,%f)'
          % (i,nFrames,time.time()-t1,acc_top1/(i+1),acc_top5/(i+1), acc_top10/(i+1)))
```

The final part of the loop first saves the `prediction` array in hdf5 format. The softmax operation is used on the output providing class probabilities for each frame. The line `prediction = np.sum(np.log(prediction),axis=0)` is a naive way of calculating log(P(Y|X)) and choosing the most likely class by assuming each frame is independent of the other frames (although they're not independent). `prediction` can also be used to get the 10 most likely classes to calculate the different accuracies.

```
number_of_examples = np.sum(confusion_matrix,axis=1)
for i in range(NUM_CLASSES):
    confusion_matrix[i,:] = confusion_matrix[i,:]/np.sum(confusion_matrix[i,:])

results = np.diag(confusion_matrix)
indices = np.argsort(results)

sorted_list = np.asarray(class_list)
sorted_list = sorted_list[indices]
sorted_results = results[indices]

for i in range(NUM_CLASSES):
    print(sorted_list[i],sorted_results[i],number_of_examples[indices[i]])

np.save('single_frame_confusion_matrix.npy',confusion_matrix)
```

After the loop, the confusion matrix can be converted to confusion probabilities. The diagonal elements will say how often a particular class is correctly identified. The code above sorts these values and prints them all out. The saved `predictions` and `confusion_matrix` will be used later in the assignment.

Part 2 - Sequence Model

Although the single frame model achieves around 73%-75% classification accuracy for a single frame, it above should achieve between 77%-79% after combining the single frame predictions over the whole video. As mentioned above, simply averaging these predictions is a very naive way of classifying sequences. This is similar to the Bag of Words model from the NLP assignment. There are many ways to combine this information in a more intelligent way.

There are many ways to utilize the temporal information. All of the papers in the introduction essentially explore these different techniques. Part 2 of the assignment will do this by using 3D convolutions. 3D convolutions are conceptually the exact same as 2D convolutions except now they also operate over the temporal dimension (sliding window over the frames as well as the image).

The model already overfits on single frames alone. If you were to train a 3D convolutional network from scratch on UCF-101, it severely overfits and has extremely low performance. The Kinetics dataset is a much larger action recognition dataset released more recently than UCF-101. The link above goes to the Kinetics-600 dataset (500,000 videos of 600 various actions). We will use a a 3D ResNet-50 model pretrained on the Kinetics-400 dataset (300,000 videos of 400 various actions) from here. This pretrained model is located in the class directory `/projects/training/bayw/hdf5/UCF-101-hdf5` on BlueWaters.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import math
from functools import partial

__all__ = [
    'ResNet', 'resnet10', 'resnet18', 'resnet34', 'resnet50', 'resnet101',
    'resnet152', 'resnet200'
]


def conv3x3x3(in_planes, out_planes, stride=1):
    # 3x3x3 convolution with padding
    return nn.Conv3d(
        in_planes,
        out_planes,
        kernel_size=3,
        stride=stride,
        padding=1,
        bias=False)


def downsample_basic_block(x, planes, stride):
    out = F.avg_pool3d(x, kernel_size=1, stride=stride)
    zero_pads = torch.Tensor(
        out.size(0), planes - out.size(1), out.size(2), out.size(3),
        out.size(4)).zero_()
    if isinstance(out.data, torch.cuda.FloatTensor):
        zero_pads = zero_pads.cuda()

    out = Variable(torch.cat([out.data, zero_pads], dim=1))

    return out


class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(BasicBlock, self).__init__()
        self.conv1 = conv3x3x3(inplanes, planes, stride)
        self.bn1 = nn.BatchNorm3d(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3x3(planes, planes)
```

```
            self.bn2 = nn.BatchNorm3d(planes)
            self.downsample = downsample
            self.stride = stride

        def forward(self, x):
            residual = x

            out = self.conv1(x)
            out = self.bn1(out)
            out = self.relu(out)

            out = self.conv2(out)
            out = self.bn2(out)

            if self.downsample is not None:
                residual = self.downsample(x)

            out += residual
            out = self.relu(out)

            return out


    class Bottleneck(nn.Module):
        expansion = 4

        def __init__(self, inplanes, planes, stride=1, downsample=None):
            super(Bottleneck, self).__init__()
            self.conv1 = nn.Conv3d(inplanes, planes, kernel_size=1, bias=False)
            self.bn1 = nn.BatchNorm3d(planes)
            self.conv2 = nn.Conv3d(
                planes, planes, kernel_size=3, stride=stride, padding=1, bias=False)
            self.bn2 = nn.BatchNorm3d(planes)
            self.conv3 = nn.Conv3d(planes, planes * 4, kernel_size=1, bias=False)
            self.bn3 = nn.BatchNorm3d(planes * 4)
            self.relu = nn.ReLU(inplace=True)
            self.downsample = downsample
            self.stride = stride

        def forward(self, x):
            residual = x

            out = self.conv1(x)
            out = self.bn1(out)
            out = self.relu(out)

            out = self.conv2(out)
            out = self.bn2(out)
            out = self.relu(out)

            out = self.conv3(out)
            out = self.bn3(out)

            if self.downsample is not None:
                residual = self.downsample(x)

            out += residual
            out = self.relu(out)

            return out


    class ResNet(nn.Module):

        def __init__(self,
                     block,
                     layers,
                     sample_size,
                     sample_duration,
                     shortcut_type='B',
```

```
                 num_classes=400):
        self.inplanes = 64
        super(ResNet, self).__init__()
        self.conv1 = nn.Conv3d(
            3,
            64,
            kernel_size=7,
            stride=(1, 2, 2),
            padding=(3, 3, 3),
            bias=False)
        self.bn1 = nn.BatchNorm3d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool3d(kernel_size=(3, 3, 3), stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, layers[0], shortcut_type)
        self.layer2 = self._make_layer(
            block, 128, layers[1], shortcut_type, stride=2)
        self.layer3 = self._make_layer(
            block, 256, layers[2], shortcut_type, stride=2)
        self.layer4 = self._make_layer(
            block, 512, layers[3], shortcut_type, stride=2)
        last_duration = int(math.ceil(sample_duration / 16))
        last_size = int(math.ceil(sample_size / 32))
        self.avgpool = nn.AvgPool3d(
            (last_duration, last_size, last_size), stride=1)
        self.fc = nn.Linear(512 * block.expansion, num_classes)

        for m in self.modules():
            if isinstance(m, nn.Conv3d):
                m.weight = nn.init.kaiming_normal(m.weight, mode='fan_out')
            elif isinstance(m, nn.BatchNorm3d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()

    def _make_layer(self, block, planes, blocks, shortcut_type, stride=1):
        downsample = None
        if stride != 1 or self.inplanes != planes * block.expansion:
            if shortcut_type == 'A':
                downsample = partial(
                    downsample_basic_block,
                    planes=planes * block.expansion,
                    stride=stride)
            else:
                downsample = nn.Sequential(
                    nn.Conv3d(
                        self.inplanes,
                        planes * block.expansion,
                        kernel_size=1,
                        stride=stride,
                        bias=False), nn.BatchNorm3d(planes * block.expansion))

        layers = []
        layers.append(block(self.inplanes, planes, stride, downsample))
        self.inplanes = planes * block.expansion
        for i in range(1, blocks):
            layers.append(block(self.inplanes, planes))

        return nn.Sequential(*layers)

    def forward(self, x):
        with torch.no_grad():
            h = self.conv1(x)
            h = self.bn1(h)
            h = self.relu(h)
            h = self.maxpool(h)

            h = self.layer1(h)
            h = self.layer2(h)
            h = self.layer3(h)
            h = self.layer4(h)
```

```
            h = self.avgpool(h)

            h = h.view(h.size(0), -1)
            h = self.fc(h)

            return h


    def get_fine_tuning_parameters(model, ft_begin_index):
        if ft_begin_index == 0:
            return model.parameters()

        ft_module_names = []
        for i in range(ft_begin_index, 5):
            ft_module_names.append('layer{}'.format(i))
        ft_module_names.append('fc')

        parameters = []
        for k, v in model.named_parameters():
            for ft_module in ft_module_names:
                if ft_module in k:
                    parameters.append({'params': v})
                    break
            else:
                parameters.append({'params': v, 'lr': 0.0})

        return parameters


    def resnet10(**kwargs):
        """Constructs a ResNet-18 model.
        """
        model = ResNet(BasicBlock, [1, 1, 1, 1], **kwargs)
        return model


    def resnet18(**kwargs):
        """Constructs a ResNet-18 model.
        """
        model = ResNet(BasicBlock, [2, 2, 2, 2], **kwargs)
        return model


    def resnet34(**kwargs):
        """Constructs a ResNet-34 model.
        """
        model = ResNet(BasicBlock, [3, 4, 6, 3], **kwargs)
        return model


    def resnet50(**kwargs):
        """Constructs a ResNet-50 model.
        """
        model = ResNet(Bottleneck, [3, 4, 6, 3], **kwargs)
        return model


    def resnet101(**kwargs):
        """Constructs a ResNet-101 model.
        """
        model = ResNet(Bottleneck, [3, 4, 23, 3], **kwargs)
        return model


    def resnet152(**kwargs):
        """Constructs a ResNet-101 model.
        """
        model = ResNet(Bottleneck, [3, 8, 36, 3], **kwargs)
        return model
```

```
def resnet200(**kwargs):
    """Constructs a ResNet-101 model.
    """
    model = ResNet(Bottleneck, [3, 24, 36, 3], **kwargs)
    return model
```

First create a new file `resnet_3d.py` with the above model definition. Next create a new python script for training the 3D convolution model.

```
import numpy as np
import os
import sys
import time

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
#from torchvision import datasets, transforms
from torch.autograd import Variable
import torch.distributed as dist
import torchvision

from helperFunctions import getUCF101
from helperFunctions import loadSequence
import resnet_3d

import h5py
import cv2

from multiprocessing import Pool


IMAGE_SIZE = 224
NUM_CLASSES = 101
batch_size = 32
lr = 0.0001
num_of_epochs = 10


data_directory = '/projects/training/bayw/hdf5/UCF-101-hdf5'
class_list, train, test = getUCF101(base_directory = data_directory)
```

This is very similar to part 1. Notice we importing `loadSequence()` this time as well as the model definition.

```
model =  resnet_3d.resnet50(sample_size=IMAGE_SIZE, sample_duration=16)
pretrained = torch.load(data_directory + 'resnet-50-kinetics.pth')
keys = [k for k,v in pretrained['state_dict'].items()]
pretrained_state_dict = {k[7:]: v.cpu() for k, v in pretrained['state_dict'].items()}
model.load_state_dict(pretrained_state_dict)
model.fc = nn.Linear(model.fc.weight.shape[1],NUM_CLASSES)
```

When creating the `model` object, we are defining the `sample_duration=16` . That is, we are going to be training on subsequences of 16 frames. As the input passes through the network, there will be max pooling operations over the spatial and temporal dimensions as well as strided 3D convolutions which will reduce the spatial and temporal dimensions by a factor of 2. By the time the final layer is processed, the temporal dimension has collapsed meaning you have a single vector output for all 16 frames.

The state dictionary for the pretrained model is loaded into `pretrained` . The names of the of layers are used to place the weights into the model we defined. Redefine the fully connected layer once

again such that it has the appropriate number of outputs.

```
for param in model.parameters():
    param.requires_grad_(False)

# for param in model.conv1.parameters():
#     param.requires_grad_(True)
# for param in model.bn1.parameters():
#     param.requires_grad_(True)
# for param in model.layer1.parameters():
#     param.requires_grad_(True)
# for param in model.layer2.parameters():
#     param.requires_grad_(True)
# for param in model.layer3.parameters():
#     param.requires_grad_(True)
for param in model.layer4[0].parameters():
    param.requires_grad_(True)
for param in model.fc.parameters():
    param.requires_grad_(True)

params = []
# for param in model.conv1.parameters():
#     params.append(param)
# for param in model.bn1.parameters():
#     params.append(param)
# for param in model.layer1.parameters():
#     params.append(param)
# for param in model.layer2.parameters():
#     params.append(param)
# for param in model.layer3.parameters():
#     params.append(param)
for param in model.layer4[0].parameters():
    params.append(param)
for param in model.fc.parameters():
    params.append(param)


model.cuda()

optimizer = optim.Adam(params,lr=lr)

criterion = nn.CrossEntropyLoss()

pool_threads = Pool(8,maxtasksperchild=200)
```

This is very similar to the single frame model as in we will only fine-tune the output layer and the last residual block. Note that `model.layer4` actually has three residual blocks. However, BlueWaters does not have a large enough GPU to handle the blocks `model.layer4[1]` and `model.layer4[2]`. The largest attainable batch size with these layers included is 1 which is not enough to train the model due to batch normalization. Instead, we will simply ignore these last two blocks and directly send the output of `model.layer4[0]` into the fully connected layer. How this is done will be shown in the training loop.

```
for epoch in range(0,num_of_epochs):

    ###### TRAIN
    train_accu = []
    model.train()
    random_indices = np.random.permutation(len(train[0]))
    start_time = time.time()
    for i in range(0, len(train[0])-batch_size,batch_size):

        augment = True
        video_list = [(train[0][k],augment)
                      for k in random_indices[i:(batch_size+i)]]
        data = pool_threads.map(loadSequence,video_list)
```

```
        next_batch = 0
        for video in data:
            if video.size==0: # there was an exception, skip this
                next_batch = 1
        if(next_batch==1):
            continue

        x = np.asarray(data,dtype=np.float32)
        x = Variable(torch.FloatTensor(x),requires_grad=False).cuda().contiguous()

        y = train[1][random_indices[i:(batch_size+i)]]
        y = torch.from_numpy(y).cuda()

        with torch.no_grad():
            h = model.conv1(x)
            h = model.bn1(h)
            h = model.relu(h)
            h = model.maxpool(h)

            h = model.layer1(h)
            h = model.layer2(h)
            h = model.layer3(h)
        h = model.layer4[0](h)

        h = model.avgpool(h)

        h = h.view(h.size(0), -1)
        output = model.fc(h)

        # output = model(x)

        loss = criterion(output, y)
        optimizer.zero_grad()

        loss.backward()
        optimizer.step()

        prediction = output.data.max(1)[1]
        accuracy = ( float( prediction.eq(y.data).sum() ) /float(batch_size))*100.0
        if(epoch==0):
            print(i,accuracy)
        train_accu.append(accuracy)
    accuracy_epoch = np.mean(train_accu)
    print(epoch, accuracy_epoch,time.time()-start_time)

    ##### TEST
    model.eval()
    test_accu = []
    random_indices = np.random.permutation(len(test[0]))
    t1 = time.time()
    for i in range(0,len(test[0])-batch_size,batch_size):
        augment = False
        video_list = [(test[0][k],augment)
                      for k in random_indices[i:(batch_size+i)]]
        data = pool_threads.map(loadSequence,video_list)

        next_batch = 0
        for video in data:
            if video.size==0: # there was an exception, skip this batch
                next_batch = 1
        if(next_batch==1):
            continue

        x = np.asarray(data,dtype=np.float32)
        x = Variable(torch.FloatTensor(x)).cuda().contiguous()

        y = test[1][random_indices[i:(batch_size+i)]]
        y = torch.from_numpy(y).cuda()
```

```
            # with torch.no_grad():
            #     output = model(x)
            with torch.no_grad():
                h = model.conv1(x)
                h = model.bn1(h)
                h = model.relu(h)
                h = model.maxpool(h)

                h = model.layer1(h)
                h = model.layer2(h)
                h = model.layer3(h)
                h = model.layer4[0](h)
                # h = model.layer4[1](h)

                h = model.avgpool(h)

                h = h.view(h.size(0), -1)
                output = model.fc(h)

            prediction = output.data.max(1)[1]
            accuracy = ( float( prediction.eq(y.data).sum() ) /float(batch_size))*100.0
            test_accu.append(accuracy)
            accuracy_test = np.mean(test_accu)

        print('Testing',accuracy_test,time.time()-t1)

    torch.save(model,'3d_resnet.model')
    pool_threads.close()
    pool_threads.terminate()
```

There are really only two differences when compared to the single frame training script.

```
augment = True
video_list = [(train[0][k],augment)
               for k in random_indices[i:(batch_size+i)]]
data = pool_threads.map(loadSequence,video_list)
```

Here we are using the function `loadSequence()` instead of `loadFrame()`. This will grab a subsequence of 16 frames from the video for training/testing. The input `x` will now be size `[batch_size,3,16,224,224]`.

```
with torch.no_grad():
    h = model.conv1(x)
    h = model.bn1(h)
    h = model.relu(h)
    h = model.maxpool(h)

    h = model.layer1(h)
    h = model.layer2(h)
    h = model.layer3(h)
h = model.layer4[0](h)

h = model.avgpool(h)

h = h.view(h.size(0), -1)
output = model.fc(h)
```

The above code is how to manage avoiding the use of `model.layer4[1]` and `model.layer4[2]` to save GPU memory. Instead of calling the `forward()` function of `model`, we can manually apply each layer. The first portion of the network can be run under `with torch.no_grad():` since these weights will not be trained.

The last two residual blocks in `model.layer4` are already trained and provide valuable features. If it was possible, they should both be used. However, the model still performs very well even without

these two layers. During the training loop, the training accuracy will once again max out at around 100% and the testing accuracy should be around 81%-83%.

Testing

Once again, we can achieve better performance on the test dataset by using the full video instead of just 16 frames. Use the code provided in part 1 for testing as a starting point to create similar code for testing the 3D ResNet on the full video sequences. There can be a couple of approaches.

- Process subsequences of length 16 and average the final output. This is the most basic way and easiest to implement but not the best way. Notice there is a difference between processing every 16 frames compared with processing every subsequence of length 16. There are overlapping subsequences. If you are unsure of how to do the next option, it may be smart to start here just to get something working to see what to expect.
- Pass the full sequence in as input. When we defined the model, we set `sample_duration=16`. This is the minimum length for training but not the max length. With 16 frames, the temporal dimension has been reduced to size 1. That is, the output after `model.layer4[0]` is `[batch_size,2048,1,7,7]` where 2048 is the number of channels, 1 is the temporal dimension, and 7x7 is the spatial dimension. By specifying `sample_duration=16`, `model.avgpool()` is created with the knowledge that average pooling will need to be performed over the 1x7x7 dimensions. If a sequence of length 32 was passed into the network, the output of `model.layer4[0]` would be `[batch_size,2048,2,7,7]`. If average pooling is performed over the 2x7x7 dimensions, then this can be passed through the `model.fc` layer to get a single prediction for the entire sequence of length 32. For each video, perform the appropriately sized averaging pooling such that the full sequence can be passed through the network. You may want to look at the model definition `resnet_3d.py` for ideas.

Save the output for each video (this should be a single prediction as opposed to a prediction for each frame). Also save the confusion matrix.

Turn In

There are three sets of results for comparison: 1.) single-frame model, 2.) 3D model, 3.) combined output of the two models. For each of these three, report the following:

- `(top1_accuracy,top5_accuracy,top10_accuracy)`: Did the results improve after combining the outputs?
- Use the confusion matrices to get the 10 classes with the highest performance and the 10 classes with the lowest performance: Are there differences/similarities? Can anything be said about whether particular action classes are discriminated more by spatial information versus temporal information?
- Use the confusion matrices to get the 10 most confused classes. That is, which off-diagonal elements of the confusion matrix are the largest: Are there any notable examples?

Put all of the above into a report and submit as a pdf. Also zip all of the code (not the models, predictions or dataset) and submit.