Behavior recognition using Kinect

Jiangjian Guo Qiming Liu Weihao Yan

January 10, 2019

Abstract

Human action recognition has great practical value and is currently a hot research field of artificial intelligence and machine learning. Our project intends to use the appropriate method to enable the machine to recognize the human body in real time through Kinect and mark the corresponding action.

We first perform feature extraction and preprocessing on the dataset through Openpose, which converts the video samples into coordinates of key points of the human body. We then used a convolutional neural network approach to make the machine learn the characteristics of the relevant actions and achieved good results in real-time testing. In order to overcome the problems of insufficient data and recognition errors, we also built our own data sets and built a new deep convolutional network. By comparison, the test results have been greatly improved.

To improve the practical value of the project, we added the function of multi-person action recognition in real-time action recognition, and intend to add depth pictures for training in subsequent work to improve the recognition accuracy.

In this paper, we will detail the complete process and implementation of the project, and give the test results. Of course, there are still a lot of things to improve in the project, we will also introduce them and try to propose possible solutions.

Contents

1 Introduction and Motivation

Human behavior recognition is a bench of computer version, which has been a popular research area in fields of artificial intelligence and machine learning in recent years. It has so many applications such as human-computer interfaces, content-based video retrieval and public security guard etc. At the same time, behavior recognition is a very challenging task because given human action can have different meanings. Different people may also perform an action in dynamic ways. Moreover, changing environment element such as light, background, occlusion and view direction may bring much uncertainty to the recognizing processes.

Let's imagine such a situation happening in the coming future. Games will no longer be limited to computer and console games. Motion sensing games will become more popular concerning health and interaction requirement. However, good motion sensing games need an action recognition system to recognize different actions of player. It should be real-time and precise so that everyone can enjoy it. Human action recognition is also a significant part in AI assistant, which will become more and more popular in next few years. Evenmore, the recognizing system can be applied on many domains. For example, we can monitor old people's action to avoid some emergent accidents. We can also apply it on robots so that they can interact with human better. A well-performed action recognition system can profoundly change our life. Inspired by the amazing usage of action recognition, our team was thinking about building a real-time human behavior recognition system.

Our project is aimed at making computer be aware of human action and recognize what behaviors the human is performing in an effectively and efficiently way. We are required to use Microsoft Kinect, which is a series of peripheral containing RGB camera, depth camera, microphones, etc. The final project result is supposed to output the bounding box of human and related action labels in a real-time video.

In our project, we used Openpose to preprocess and do feature extration to our original dataset. Many algorithms including C3D, C2D, light flow have been tried. We also created our own dataset to have a better result of recognition. Detailed information and result of our project will be introduced in this report. Also, the follow-up research plan and some defect will also be introduced.

2 Background

2.1 Prior Work

2.1.1 Action Representation

Holistic representation methods capture the motion information of the entire human subject. Local representations only identify local regions having salient motion information.

2.1.2 Action Classification

Action classifiers can be roughly divided into the following categories: Classification, Sequential Approaches, Space-time Approaches, Partbased Approaches, Manifold Learning Approaches, Mid_Level Feature Approaches and Feature Fusion Approaches

2.1.3 Deep Network

In recent years, feature learning using deep learning techniques has been receiving increasing attention due to their ability of designing powerful features that can be generalized very well [2, 3, 5] . Recent deep networks [2, 4, 6] have achieved surprisingly high recognition performance on a variety of action datasets.

2.2 Research Challenges

Despite significant progress has been made in human action recognition and prediction, state-of-the-art algorithms still misclassify actions due to several major challenges.

2.2.1 Intra- and Inter-class Variations

People behave differently for the same actions. Videos in the same action can be captured from various viewpoints

2.2.2 Cluttered Background and Camera Motion

Most of existing activity features such as histograms of oriented gradient and interest points also encode background noise, and thus degrade the recognition performance. Camera motion is another factor that should be considered in real-world applications. Due to significant camera motion, action features cannot be accurately extracted.

2.2.3 Insufficient Annotated Data

Even though existing action recognition approaches have shown impressive performance on small-scale datasets in laboratory settings, it is really challenging to generalize them to real-world applications due to their inability of training on largescale datasets.

3 Implementation Details

3.1 Dataset

Finding a suitable dataset for our algorithm is a time-consuming task. Most datasets available on the internet are either too small or without acceptable and correct labels. We spent several days and finally decided to use FLORENCE 3D ACTIONS DATASET as our preliminary dataset. This dataset includes 9 actions: wave, drink from a bottle, answer phone, clap, tight lace, sit down, stand up, read watch, bow. Each action is performed by ten people and each person should perform an action for 2 or 3 times. Some screenshots are pictured in Figure 1.

Figure 1: Some screenshots of dataset.

Obviously, this dataset is very small but also well-labeled. We decide to use it to confirm our algorithm first, and then replace the dataset with a much bigger one. We created our own dataset during project time, related work will be introduced later in this report.

3.2 Feature Extraction Using Openpose

In FLORENCE 3D ACTIONS DATASET, each sample is in video form, each video has 10 to 25 frames, each frame has 600*480 pixels, and each pixel has three channels (RGB). The total dimension of one sample is extremely large, we cannot get a good result in just 215 samples, which is the main challenge at the beginning of our project. Clearly, we need dimensionality reduction, feature extraction and data preprocessing.

We decided to use Openpose [7, 8, 9, 10] as a tool to help us with this essential work. Openpose represents a real-time multi-person system to detect human body, hand, facial, and foot key points on single images. It used deep net method (VGG, Resnet, Mobilenet, etc.) and Part Affinity Fields (PAFs) to estimate one's body key point. To learn more about openpose, you can visit its website on Github.

Note that we only need to load the pretrained model in Openpose to estimate key point of body in a picture. Openpose offered several models which differs in the recognizing speed, recognizing accuracy and the amount of key point. To fit our project task, we decide to use the MPI model. This model will output 15 key points of one's body. The test output is showed in Figure 2. When the model cannot recognize key parts of body, it will return a None type variable, which represents the part is missing.

Clearly seen from the test result, Openpose can transfer a high dimension picture to a series of key point positions, which in very low dimension. Furthermore, Openpose also helps us to extract features from the original graph, so that we can represent actions the human is playing with some point positions. Learning the moving law of key points is also learning how the actual human behavior looks like.

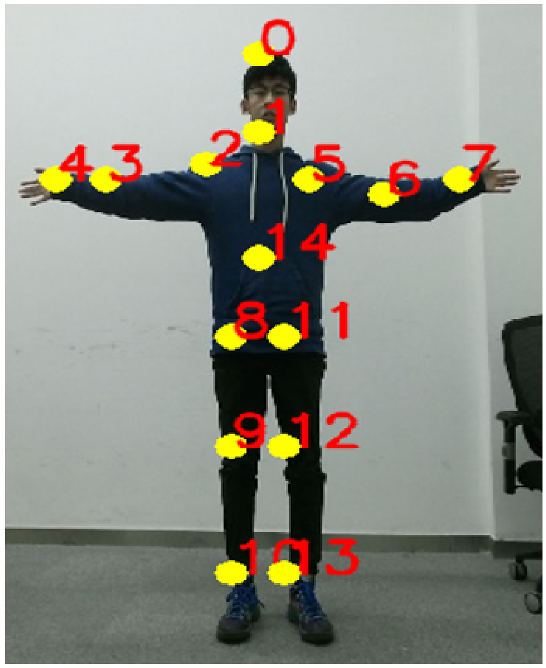

Figure 2: Some screenshots of dataset.

In practice, we divide a sample video into 8 frames, each frame can have X and Y position of 15 key points using Openpose. We spliced the position of key points in the order of frame occurred. In this way we got the preprocess new dataset.

3.3 Introduction to Deep Learning Network

Considering our data's shape, there are 8 frames in one action and in each frame there are 15 key points which have x and y two dimensions. Aftering reading this reference[1], we dicide to use 2D Convolution. Specifically, our first net is shown in Figure 3.

In the net, x and y are two different channels while 8 frames and 15 key points are two dimensions to be convoluted. The convolution structure is shown in Figure 4.

Running the preprocessed dataset on this net, we finally got around 80% accuracy, detailed training process and result will be introduce at the Experiment part.

3.4 New Dataset: Action-209

In practice, we found that the model is not good enough because we still get some errors in real-time recognition. We tuned the parameters to improve the result but the test accuracy is always around 80%, which means the model need better generalization ability. We believe it is because the datasets we use is too small and monotonous. So we decided to build our own datasets. We call it Action-209 since it is collected in Lizhengdao library room 209.

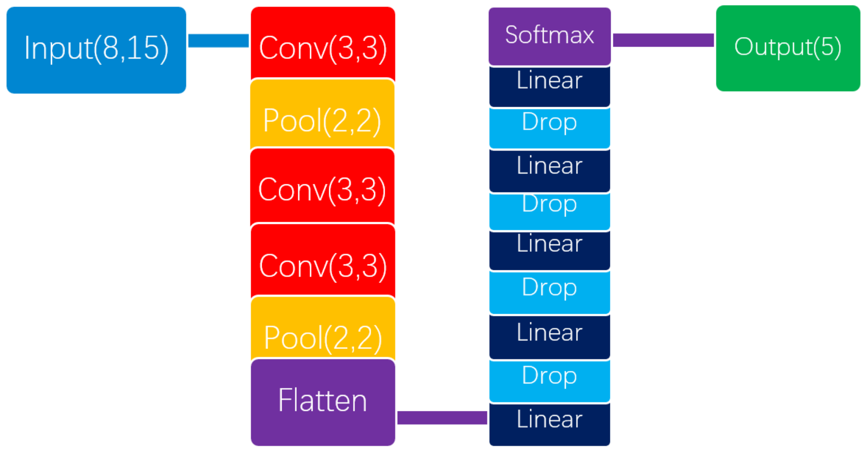

Figure 3: The structure of our first net

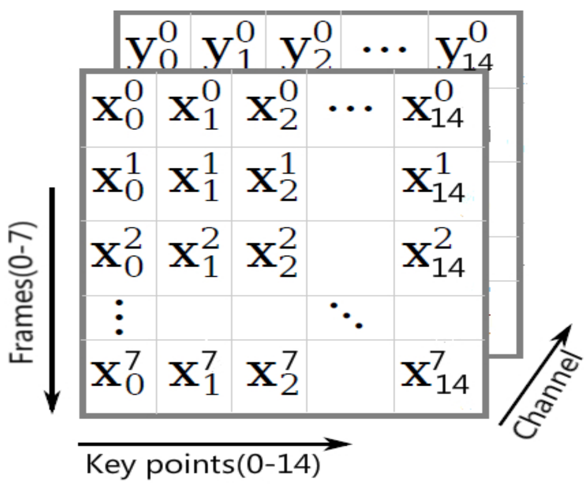

Figure 4: The convolution structure

We use the function of Pykinect API to catch real-time RGB and depth frames. Sampling frequency is fixed so that trained model will have a stable performance.

We designed the dataset according to the shortcomings of the previous one.

FLORENCE 3D ACTIONS DATASET	Action-209
Only 5 high-quality actions	Expanded to 9 actions
25 series for an action	Nearly 240 series of images
Always from starts to ends	Might start at any time
Only has 8 frames	Collect 16 frames in a row
Low frequency	More detailed information
Always act in the center	Collect images of all positions
Similar speed when acting	Different speed when acting

Table 1: The tunning process of parameters

We collect 240 samples for one action, each of them has 16 frames. It is much larger and better than the previous one and we did get a more precise model trained with it.

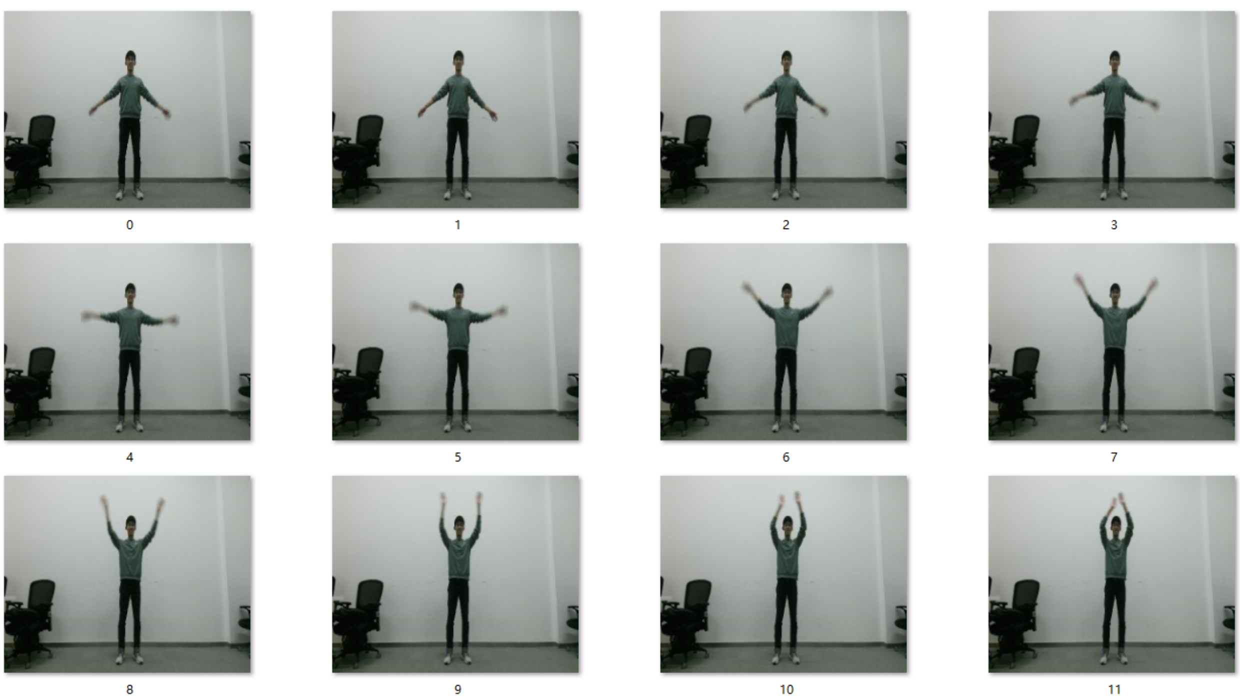

Figure 5: Action-209 dataset created by our team.

The datasets are collected in a tiny room containing data of only 3 group members so it still need to be improved. We will talk about it in Discussion part.

3.5 New Deep Network Based on Actions-209

Owing to more data, more frames and more action types, we need to update our fist net to a more complex one. After adjusting the structure, the new net is is shown in Figure 6.

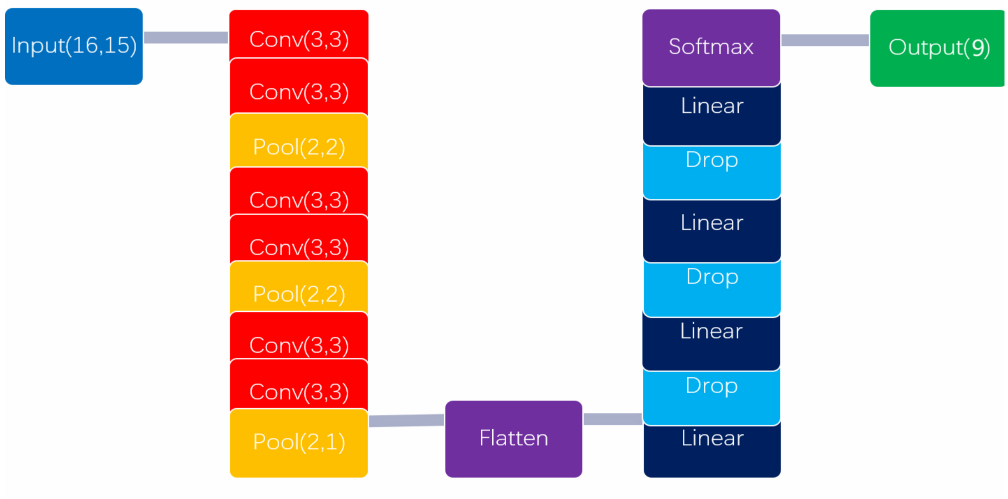

Figure 6: The structure of our second net

The convolution structure is updated to picture shown in Figure 7.

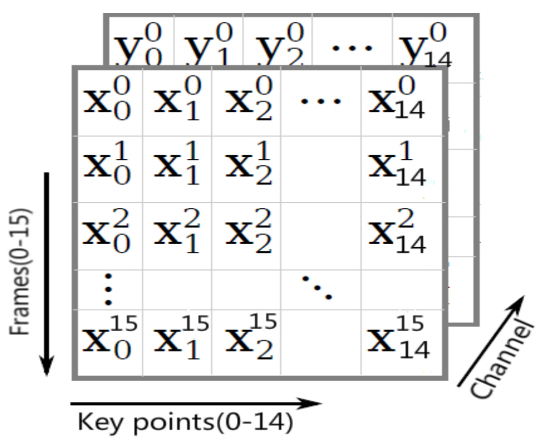

Figure 7: The convolution structure

Running the preprocessed dataset on this net, we finally got around 95% accuracy, detailed training process and result will be introduce at the Experiment part.

4 Experiments

4.1 Real-time Video Sampling

We planned to put the model into practical applications, which means we want to do real-time recognition with frames from Kinect rather than a preprocessed video. Fortunately, we found an API called pykinect2, which can catch RGB-D frame and joints coordinates instantly. So, while running the demo, we can have real-time information of the human in front of Kinect.

In real-time application, we keep pace with the sampling of dataset to get instant coordinates. We collect information of 16 frames and preprocess it to fit the model. The network then will give us a prediction label which is the action label we will update with.

Basically, we got a pretty good recognition system which can do well in real-time running. It can accurately distinguish 9 different actions right now.

What's more, we make our idea of two-people action recognition come true. When there are two people in front of Kinect doing actions, our algorithms can recognize them respectively. In general, we can increase the number of people to 6 at most.

But it still makes some mistakes during testing. We will talk about it in Discussion.

4.2 Parameters Tunning

4.2.1 The Tunning Process and Experiment Result of First Net

The main tunning parameters of our net are max_iterations, batch_size and learning_rate. We tried many conbinations of parameters, which has been listed in Table 2.

max_iterations	batch_size	learning_rate	train_accuracy(%)	test_accuracy(%)
50	4	0.001	64.00	69.47
100	4	0.001	82.67	76.84
150	4	0.0005	95.20	74.73
150	5	0.0005	96.53	81.05
150	10	0.0005	93.87	84.21
200	10	0.0005	72.53	66.31
200	10	0.0001	66.13	52.63
200	20	0.0005	88.53	70.52
200	20	0.0001	59.46	46.32
200	15	0.0005	95.20	87.37

Table 2: The tunning process of parameters

According to Table 2, we chose max_iterations = 200, batch_size = 15 and learning_rate = 0.0005, the training process is shown in Figure 8.

From the result we know that after 200 iterations, the net's training accuracy is above 95% and testing accuracy is more than 85%, which seems a good result. However, due to

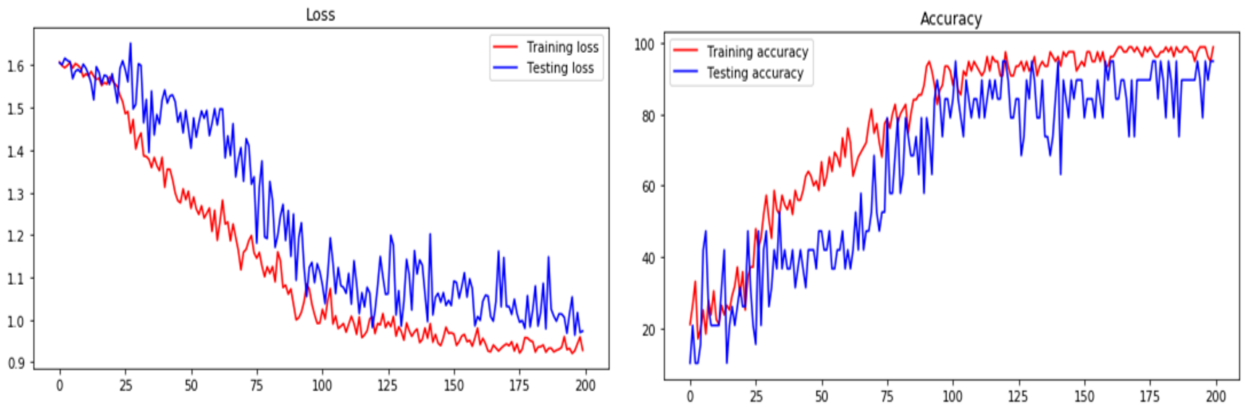

Figure 8: The variation trend of loss and accuracy of final version in the training process

small size of dataset and randomness of the training process, the result of training varies from time to time. Sometimes the testing accuracy is below 75%, which is not good enough.

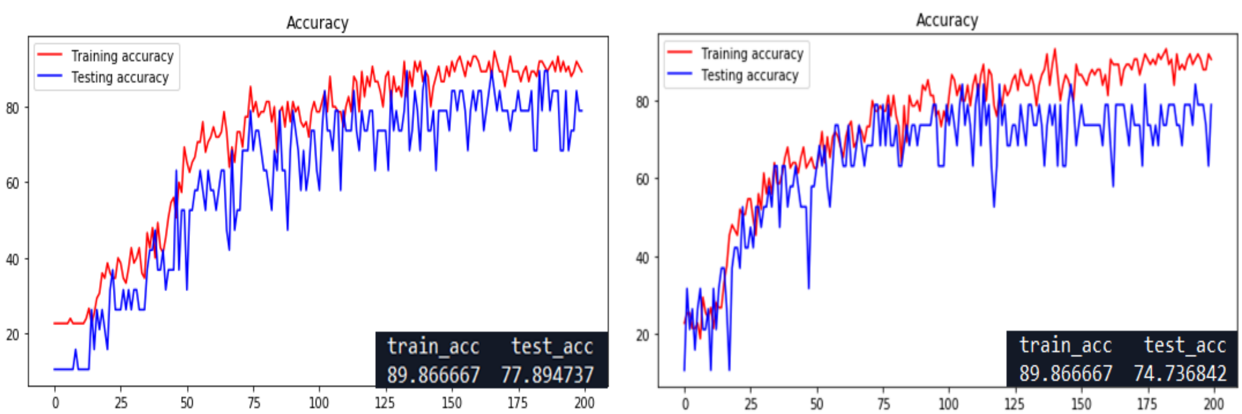

Figure 9: The accuracy varies from time to time

4.2.2 The Tunning Process and Experiment Result of Second Net

We tried many conbinations of parameters, which has been listed in Table 3.

Finally we chose max_iterations = 300, batch_size = 4 and learning_rate = 0.00005, the training process is shown in Figure 10.

From the result we can see that after 300 iterations, both the training accuracy and testing accuracy of our second net are higher than 90% which shows good performance and the curves convege more stablely. What's more, after running the traning process for several times, the acuracies are always higher than 90%.

We also tried to use RandomForest to get the prediction model. After tunning the hyper-parameters in RanomForest Classifier, the final result is shown in Figure 11:

max_iterations	batch_size	learning_rate	train_accuracy(%)	test_accuracy(%)
100	4	0.001	15.65	15.64
100	4	0.0005	17.47	17.04
150	4	0.0005	76.66	86.59
150	4	0.0001	92.73	93.30
200	10	0.0001	94.56	93.97
300	10	0.00005	86.44	85.47
300	10	0.00001	65.13	67.60
300	4	0.00005	96.51	95.81

Table 3: The tunning process of parameters

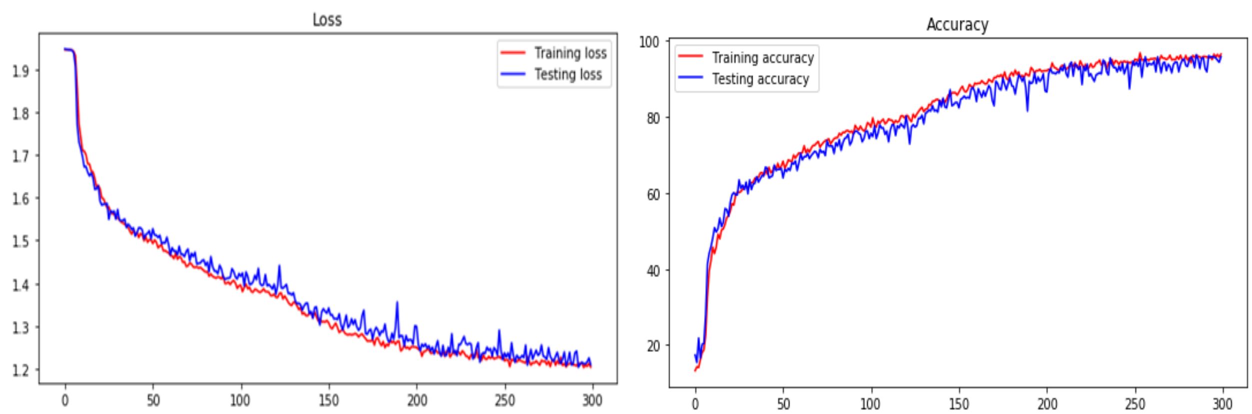

Figure 10: The variation trend of loss and accuracy of final version in the training process

```
RandomForestClassifier
When using RandomForest:
When k =  10
When use cross_val_score:
Accuracy: 0.98687(+/-)0.00003
When use cross_val_predict:
MSE: 0.09965
total time: 8.845
```

Figure 11: The training result using Random Forest

From the result in Figure 11 we can see that when use 10-fold cross validation, the accuracy is nearly 98% which is higher than our second net. However, when we used the high accuracy new net to do real time prediction, it performed badly so we finally didn't use the new net.

5 Source code

5.1 Dataset Preprocessing Using Openpose

Python file `preprocessing_using_openpose.py` is used to transfer the original video into a series of body key points. This file first transfer the video into frames and then input each frame into Openpose pretrained model to get the position of key points.

Note that if you want to run this file, you first need to have the pretrained model, which will be handed in to TA. As for our own dataset, the main part of Openpose is the same, so we won't list it here.

```
import cv2
import os
import random
import numpy as np
import scipy.io as io
import matplotlib.pyplot as plt

def get_path(path):
    path_list = os.listdir(path)
    path_list.sort()
    full_path_list = []
    for filename in path_list:
        full_path_list.append(os.path.join(path,filename))
    return full_path_list, len(full_path_list)


def get_frame_list(frame_number, get_number):
    if frame_number < get_number:
        flist = list(range(frame_number))
        for k in range(get_number - frame_number):
            flist.append(frame_number - 1)
    elif frame_number < 1.3 * get_number:
        flist = []
        for k in range(get_number):
            flist.insert(0, range(frame_number)[-1 - k])
    else:
        frame_list = list(range(round(frame_number * 0.18), frame_number))
        flist = random.sample(frame_list, get_number)
        flist.sort()

    return flist

# cut the video into frames and implement openpose
def cut_video(path, path_folder_naming, arr_mat, arr_lab):
    protoFile = r'C:\Users\Administrator\Desktop\AI_Project\openpose-master\models\pose\mpi\pose_deploy_linevec.prototxt'
    weightsFile = r'C:\Users\Administrator\Desktop\AI_Project\openpose-master\models\pose\mpi\pose_iter_160000.caffemodel'
    net = cv2.dnn.readNetFromCaffe(protoFile, weightsFile)
    temp_list_x = []
    temp_list_y = []

    cap = cv2.VideoCapture(path)
    category = eval(path[-5])
    path_folder_naming[category - 1] += 1
    naming_num = path_folder_naming[category - 1]
    writing_folder = 'C:\\Users\\Administrator\\Desktop\\AI_Project\\dataset\\' + str(category) + '\\' + str(naming_num) + '\\'
    os.mkdir(writing_folder)

```

```
    total_frame = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    get_number = 8
    frame_list = get_frame_list(total_frame, get_number)


    if cap.isOpened():
        print('video successfully opened')
        success = True

    for i in range(get_number):
        cap.set(cv2.CAP_PROP_POS_FRAMES, frame_list[i])
        success, frame = cap.read()

        print('Reading a new frame: ', success)
        if success:
            cv2.imwrite(writing_folder + "frame" + "_%d.jpg" % i, frame, [int(cv2.
                IMWRITE_JPEG_QUALITY), 100])
            temp_list_x, temp_list_y = get_key_point_from_frame(frame, net, temp_list_x,
                temp_list_y)

    temp_list_x.extend(temp_list_y)
    arr_mat.append(temp_list_x)
    arr_lab.append(category)

    cap.release()
    return path_folder_naming, arr_mat, arr_lab


def get_key_point_from_frame(frame, net, temp_list_x, temp_list_y):

    # Specify number of points in the model
    nPoints = 15
    im = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    inWidth = im.shape[1]
    inHeight = im.shape[0]

    # Convert image to blob
    netInputSize = (368, 368)
    inpBlob = cv2.dnn.blobFromImage(im, 1.0 / 255, netInputSize, (0, 0, 0), swapRB=True,
        crop=False)
    net.setInput(inpBlob)

    # Run Inference (forward pass)
    output = net.forward()
    scaleX = float(inWidth) / output.shape[3]
    scaleY = float(inHeight) / output.shape[2]

    points = []
    # Confidence treshold
    threshold = 0.1

    for i in range(nPoints):
        probMap = output[0, i, :, :]
        minVal, prob, minLoc, point = cv2.minMaxLoc(probMap)

        # Scale the point to fit on the original image
        x = scaleX * point[0]
        y = scaleY * point[1]

        if prob > threshold :
            points.append((int(x), int(y)))
            temp_list_x.append(int(x))
            temp_list_y.append(int(y))
        else :
            points.append(None)
```

```python
            temp_list_x.append(None)
            temp_list_y.append(None)

    return temp_list_x, temp_list_y

if __name__ == '__main__':
    folder_path = r'C:\Users\Administrator\Desktop\AI_Project\dataset\Florence_3d_actions'
    video_path_list, video_num = get_path(folder_path)
    path_folder_naming = [0, 0, 0, 0, 0, 0, 0, 0, 0]
    arr_mat = []
    arr_lab = []

    for num in range(video_num):
        print('Now slice video', num + 1)
        path_folder_naming, arr_mat, arr_lab = cut_video(video_path_list[num],
            path_folder_naming, arr_mat, arr_lab)
        print(path_folder_naming)

        if num == 100:
            # print(arr_mat)
            print(np.array(arr_mat).shape)
            print(np.array(arr_lab).shape)

            np.save('sample_' + str(num) + '.npy', np.array(arr_mat))
            np.save('label_' + str(num) + '.npy', np.array(arr_lab))

    arr_mat = np.array(arr_mat)
    arr_lab = np.array(arr_lab)
    np.save('sample.npy', arr_mat)
    np.save('label.npy', arr_lab)
```

5.2 Training Process

Python file `C2D_model.py` is the code that we construct the 2D convolution training net.

The input shape of the net is (2,16,15) where 2 is the number of channels, 16 is the number of frames and 15 is the number of key points. The output is an array with 9 probs where each stands for the prediction probability of relative action.

```python
import torch.nn as nn

class C2D(nn.Module):

    def __init__(self):
        super(C2D, self).__init__()

        #input shape: (16, 15)
        self.conv1 = nn.Conv2d(2,  4, kernel_size=(3,3), padding=(1,1), stride=(1,1))#(16,
            15)
        self.conv2 = nn.Conv2d(4,  8, kernel_size=(3,3), padding=(1,1), stride=(1,1))#(16,
            15)
        self.pool1 = nn.MaxPool2d(kernel_size=(2,2),stride=(2,2))#(8, 7)

        self.conv3 = nn.Conv2d(8, 12, kernel_size=(3,3), padding=(1,1), stride=(1,1))#(8, 7)
        self.conv4 = nn.Conv2d(12,16, kernel_size=(3,3), padding=(1,1), stride=(1,1))#(8, 7)
        self.pool2 = nn.MaxPool2d(kernel_size=(2,2),stride=(2,2))#(4, 3)

        self.conv5 = nn.Conv2d(16,20, kernel_size=(3,3), padding=(1,1), stride=(1,1))#(4, 3)
        self.conv6 = nn.Conv2d(20,24, kernel_size=(3,3), padding=(1,1), stride=(1,1))#(4, 3)
        self.pool3 = nn.MaxPool2d(kernel_size=(2,1),stride=(2,1))#(2, 3)

        self.fc1 = nn.Linear(144, 72)
```

```
        self.fc2 = nn.Linear(72, 36)
        self.fc3 = nn.Linear(36,  18)
        self.fc4 = nn.Linear(18,  9)
        #output labels: 9

        self.dropout1 = nn.Dropout(p=0.1)
        self.dropout2 = nn.Dropout(p=0.2)
        self.dropout3 = nn.Dropout(p=0.3)
        self.dropout4 = nn.Dropout(p=0.4)
        self.dropout5 = nn.Dropout(p=0.5)

        self.relu = nn.ReLU()
        self.softmax = nn.Softmax()

    def forward(self, x):

        #convolution and pooling layers
        h = self.relu(self.conv1(x))
        h = self.relu(self.conv2(h))
        h = self.pool1(h)

        h = self.relu(self.conv3(h))
        h = self.relu(self.conv4(h))
        h = self.pool2(h)

        h = self.relu(self.conv5(h))
        h = self.relu(self.conv6(h))
        h = self.pool3(h)

        #flatten layer
        h = h.view(-1, 144)

        #full connection layers
        h = self.relu(self.fc1(h))
        h = self.dropout3(h)
        h = self.relu(self.fc2(h))
        h = self.dropout3(h)
        h = self.relu(self.fc3(h))
        h = self.dropout2(h)
        logits = self.fc4(h)

        probs = self.softmax(logits)

        return probs
```

Python file `predict.py` is the code that we use to train the 2D convolution net and save the model we needed.

After loading and shuffling the key points dataset, we choose former 80% samples to be training set and later 20% samples to be training set. In the training process, we tried many combinations of parameters and finally got not bad results.

```
import torch
import torch.nn as nn

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

from C2D_model import C2D

#calculate the accuracy
#y is actual label and z is prediction result(haven't changed to label)
def accuracy_cal(y, z):
```

```
    batch_acc = 0
    z_label = torch.argmax(z,1).numpy()
    for m in range(len(y)):
        if int(y[m]) == int(z_label[m]):
            batch_acc += 1
    return batch_acc

#testing set
def testing(net, testing_samples, testing_labels, loss_fn):
    test_size = 1
    test_iterate = int(len(testing_samples) / test_size)
    test_acc = 0
    test_loss = 0
    for m in range(test_iterate):
        y_pred = net(testing_samples[m*test_size:(m+1)*test_size])
        y = testing_labels[m*test_size:(m+1)*test_size]
        #calculate testing loss
        loss = loss_fn(y_pred, y.type(torch.long))
        test_loss += loss.item()
        #calculate testing accuracy
        test_acc += accuracy_cal(y,y_pred)

    return test_loss, test_acc

#training process
def training_process(net, training_samples, training_labels, testing_samples, testing_labels
    , max_iterations, batch_size, learning_rate):

    #set the parameters, batch size, max_iterations, test_size
    batch_size = batch_size
    max_iterations = max_iterations  #200
    iters_per_iterate = int(np.ceil(len(training_samples)/batch_size))

    #define loss function
    loss_fn = nn.CrossEntropyLoss(reduction = 'sum')

    #define optimizer
    optimizer = torch.optim.Adam(net.parameters(), lr = learning_rate) #0.001

    ################
    #begin training#
    ################
   #store the training and testing accuracy and loss information
    training_loss, testing_loss = [], []
    training_accuracy, testing_accuracy = [], []

    for iterate in range(max_iterations):
        iterate_loss = 0
        iterate_acc = 0
        for i in range(iters_per_iterate):
            #if come to the end
            if i == (iters_per_iterate -1) :
                x = (training_samples[i*batch_size: len(training_samples)])
                y = training_labels[i*batch_size: len(training_samples)]
            else:
                x = (training_samples[i*batch_size: (i+1)*batch_size])
                y = training_labels[i*batch_size: (i+1)*batch_size]
            z = net(x)

            #calculate the batch loss
            loss_val = loss_fn(z, (y.type(torch.long)))
            iterate_loss += loss_val.item()
            #calculate the batch accuracy
            iterate_acc += accuracy_cal(y,z)
```

```
            #update the parameters
            optimizer.zero_grad()
            loss_val.backward()
            optimizer.step()

        #update the acc and loss list
        training_loss.append(iterate_loss/len(training_samples))
        training_accuracy.append(100 * iterate_acc / len(training_samples))

        #fit testing set
        test_loss, test_acc = testing(net, testing_samples, testing_labels, loss_fn)
        testing_loss.append(test_loss / len(testing_samples))
        testing_accuracy.append(100 * test_acc / len(testing_samples))
        print(iterate, training_accuracy[-1], testing_accuracy[-1])

    #plot the images
    plot_image(training_loss, training_accuracy, testing_loss, testing_accuracy)
    #find the best model's number
    print('The best model is in:', np.argmax(np.array(testing_accuracy)))
    return np.mean(np.array(training_accuracy[-1])), np.mean(np.array(testing_accuracy[-1]))

#plot the image of training and testing accuracy and loss in the training process
def plot_image(training_loss, training_accuracy, testing_loss, testing_accuracy):

    figure, ax = plt.subplots(figsize = [9,10])
    x= range(len(training_loss))

    #plot the loss
    plt.subplot(2,1,1)
    line1, =plt.plot(x, training_loss, 'r-', label = "Training loss")
    plt.title('Loss')

    plt.subplot(2,1,1)
    line2, =plt.plot(x, testing_loss, 'b-', label = 'Testing loss')
    plt.legend(handles =[line1, line2], loc = 0)

    #plot the accuracy
    plt.subplot(2,1,2)
    line3, =plt.plot(x,training_accuracy, 'r-', label = 'Training accuracy')
    plt.title('Accuracy')

    plt.subplot(2,1,2)
    line4, =plt.plot(x,testing_accuracy, 'b-', label = 'Testing accuracy')
    plt.legend(handles = [line3, line4], loc = 0)
    #show the figure
    plt.show()



def training(net, features, labels, max_iterations, batch_size, learning_rate):

    #N: the number of samples, C: 2 channels (x,y), F: frames, H: height, actually 15 key
        points
    N, C, F, H = features.shape

    #reorder the trainig set
    np.random.seed(6)
    np.random.shuffle(features)
    np.random.seed(6)
    np.random.shuffle(labels)

    #define the number of training set and testing set
    training_num = int(N * 0.8)
    #get the training set and testing set
    training_samples = torch.Tensor(features[:training_num])
```

```
    training_labels = torch.Tensor(labels[:training_num])
    testing_samples = torch.Tensor(features[training_num:])
    testing_labels = torch.Tensor(labels[training_num:])
    #train the net
    train_acc, test_acc =  training_process(net, training_samples, training_labels,
        testing_samples, testing_labels, max_iterations,batch_size ,learning_rate)

    #show the final accuracy
    training_acc, testing_acc = [], []
    training_acc.append(train_acc)
    testing_acc.append(test_acc)
    show = pd.DataFrame(columns = ('train_acc','test_acc'))
    show['train_acc'] = training_acc
    show['test_acc'] = testing_acc
    print(show)

    #save the model
    torch.save(net, 'net.pkl')

#pre_treat the features and labels
def pre_treat(features, labels): #pre_treat the data

    #delete some labels samples(which have many None data)
    del_row = []
    for i in range(len(labels)):
        if int(labels[i]) in [4]:
            del_row.append(i)
    features = np.delete(features, del_row, 0)
    labels = np.delete(labels, del_row, 0)


    #change the remain labels to right order
    for i in range(len(labels)):
        if int(labels[i]) >= 1 and int(labels[i]) <= 3:
            labels[i] = int(labels[i]) - 1
        else:
            labels[i] = int(labels[i]) - 2

    return features, labels

#using value iteration to process None data
def filter_none_data(features):

    #get the None data's position and set them to 0 initially
    for i in range(len(features)):
        non_pos = []
        for j in range(len(features[i])):
            if str(features[i][j]) == str(None):
                non_pos.append(j)
                features[i][j] = 0

        #using 10 times value iteration to padding the missing values
        for k in range(10): #value iteration times
            for m in range(len(non_pos)):
                if non_pos[m]%15 == 0:  #begin position
                    features[i][non_pos[m]] = features[i][non_pos[m]+1]
                elif (non_pos[m]+1) % 15 == 0: #end position
                    features[i][non_pos[m]] = features[i][non_pos[m]-1]
                else:#medium position
                    features[i][non_pos[m]] = (features[i][non_pos[m]+1] + features[i][
                        non_pos[m]-1]) / 2
    return features

if __name__ == '__main__':

```

```
    #load data
    features = np.load('sample_big.npy')
    labels = np.load('label_big.npy')

    #preprocess the features and labels
    features, labels = pre_treat(features, labels)
    features = filter_none_data(features)

    #change the features to appropriate type
    features = features.astype(np.int32)

    #N,C,F,H,W, actually no W
    features = features.reshape(-1,2,16,15)

    #initial the net
    net = C2D()

    #assign the tunning parameters
    max_iterations = 300
    batch_size = 4
    learning_rate = 0.0001

    #train the net parameters
    training(net, features, labels, max_iterations, batch_size, learning_rate)
    print('max_iterations\t', 'batch_size\t', 'learning_rate')
    print(max_iterations,'\t', batch_size, '\t', learning_rate)
```

5.3 Create Dataset Action-209

Python file `record.py` is the code we used to collect our dataset Action-209. Since we have color frames of Kinect which is just the real-time images it gets, we can just store them as the datasets. So we code it based on a previous demo. The only change is that we store images in a fixed frequency and write them to disks as the game runs.

During running the game, we can get the color frames which contains rgb-d information. We reshape them to image matrixes and split rgb and depth. Both of them are resized into (600, 480) and saved. The code is listed below.

```
from pykinect2 import PyKinectV2
from pykinect2.PyKinectV2 import *
from pykinect2 import PyKinectRuntime


import ctypes
import _ctypes
import pygame
import sys
import math
import numpy as np
import cv2
import time
import os

'''
This py code is used to collect our dataset containing rgb images and depth data
it is based on a demo of drawing box (not the final version)
the function is done at # 310~340, So I only wrote notation at that part
the full notation is at done in the final virsion demo
'''

```

```
if sys.hexversion >= 0x03000000:
    import _thread as thread
else:
    import thread

# path to store the data
PATH = "E:/AI/final/PyKinect2/examples/video/10/3/"

# colors for drawing different bodies
SKELETON_COLORS = [pygame.color.THECOLORS["red"],
                   pygame.color.THECOLORS["blue"],
                   pygame.color.THECOLORS["green"],
                   pygame.color.THECOLORS["orange"],
                   pygame.color.THECOLORS["purple"],
                   pygame.color.THECOLORS["yellow"],
                   pygame.color.THECOLORS["violet"]]


class BodyGameRuntime(object):
    def __init__(self):
        pygame.init()

        # Used to manage how fast the screen updates
        self._clock = pygame.time.Clock()

        # Set the width and height of the screen [width, height]
        self._infoObject = pygame.display.Info()
        self._screen = pygame.display.set_mode((self._infoObject.current_w >> 1, self.
            _infoObject.current_h >> 1),
                                               pygame.HWSURFACE|pygame.DOUBLEBUF|pygame.
                                                   RESIZABLE, 32)

        pygame.display.set_caption("Action Recognition with Kinect")

        # Loop until the user clicks the close button.
        self._done = False

        # Used to manage how fast the screen updates
        self._clock = pygame.time.Clock()

        # Kinect runtime object, we want only color and body frames
        self._kinect = PyKinectRuntime.PyKinectRuntime(PyKinectV2.FrameSourceTypes_Color |
            PyKinectV2.FrameSourceTypes_Body)

        # back buffer surface for getting Kinect color frames, 32bit color, width and height
             equal to the Kinect color frame size
        self._frame_surface = pygame.Surface((self._kinect.color_frame_desc.Width, self.
            _kinect.color_frame_desc.Height), 0, 32)

        # here we will store skeleton data
        self._bodies = None

        # pose of user
        self._pose = "stand"

        #to get data
        self._is_get = False

        #begin record
        self._frame_num = 0
        if os.listdir(PATH) == []:
            self._group_num = 0
        else:
            n = 0
            l = os.listdir(PATH)
```

```
            for i in l:
                m = int(i)
                if m > n:
                    n = m
            self._group_num = n + 1

        self.depth = np.zeros((1,600,480))
        # data
        self._x = []
        self._y = []
        self.labels = ["wave", "applause", "sit", "stand", "watch wrist"]

        # interval
        self._inter_range = 100
        self._interval = 0

        print("model loaded")

    def draw_body_bone(self, joints, jointPoints, color, joint0, joint1):
        joint0State = joints[joint0].TrackingState;
        joint1State = joints[joint1].TrackingState;

        # both joints are not tracked
        if (joint0State == PyKinectV2.TrackingState_NotTracked) or (joint1State ==
            PyKinectV2.TrackingState_NotTracked):
            return

        # both joints are not *really* tracked
        if (joint0State == PyKinectV2.TrackingState_Inferred) and (joint1State == PyKinectV2
            .TrackingState_Inferred):
            return

        # ok, at least one is good
        start = (jointPoints[joint0].x, jointPoints[joint0].y)
        end = (jointPoints[joint1].x, jointPoints[joint1].y)

        try:
            pygame.draw.line(self._frame_surface, color, start, end, 8)
        except: # need to catch it due to possible invalid positions (with inf)
            pass

    def draw_body(self, joints, jointPoints, color):
        # Torso
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_Head,
            PyKinectV2.JointType_Neck);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_Neck,
            PyKinectV2.JointType_SpineShoulder);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_SpineShoulder,
            PyKinectV2.JointType_SpineMid);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_SpineMid,
            PyKinectV2.JointType_SpineBase);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_SpineShoulder,
            PyKinectV2.JointType_ShoulderRight);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_SpineShoulder,
            PyKinectV2.JointType_ShoulderLeft);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_SpineBase,
            PyKinectV2.JointType_HipRight);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_SpineBase,
            PyKinectV2.JointType_HipLeft);

        # Right Arm
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_ShoulderRight,
            PyKinectV2.JointType_ElbowRight);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_ElbowRight,
            PyKinectV2.JointType_WristRight);
```

```python
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_WristRight, PyKinectV2.JointType_HandRight);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_HandRight, PyKinectV2.JointType_HandTipRight);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_WristRight, PyKinectV2.JointType_ThumbRight);

        # Left Arm
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_ShoulderLeft, PyKinectV2.JointType_ElbowLeft);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_ElbowLeft, PyKinectV2.JointType_WristLeft);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_WristLeft, PyKinectV2.JointType_HandLeft);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_HandLeft, PyKinectV2.JointType_HandTipLeft);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_WristLeft, PyKinectV2.JointType_ThumbLeft);

        # Right Leg
        #self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_HipRight, PyKinectV2.JointType_KneeRight);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_KneeRight, PyKinectV2.JointType_AnkleRight);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_AnkleRight, PyKinectV2.JointType_FootRight);

        # Left Leg
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_HipLeft, PyKinectV2.JointType_KneeLeft);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_KneeLeft, PyKinectV2.JointType_AnkleLeft);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_AnkleLeft, PyKinectV2.JointType_FootLeft);


    def draw_box(self, joints, jointPoints, color):
        X = []
        Y = []
        self._interval += self._clock.get_time()
        if self._interval > self._inter_range:
            self._interval = 0
            self._is_get = True
        # [3] [20] [8] [9] [10] [4] [5] [6] [16] [17] [18] [12] [13] [14] [1]
        points = [PyKinectV2.JointType_Head, PyKinectV2.JointType_SpineShoulder, PyKinectV2.JointType_ShoulderRight,
                PyKinectV2.JointType_ElbowRight, PyKinectV2.JointType_WristRight, PyKinectV2.JointType_ShoulderLeft,
                PyKinectV2.JointType_ElbowLeft, PyKinectV2.JointType_WristLeft, PyKinectV2.JointType_HipRight,
                PyKinectV2.JointType_KneeRight, PyKinectV2.JointType_AnkleRight, PyKinectV2.JointType_HipLeft,
                PyKinectV2.JointType_KneeLeft, PyKinectV2.JointType_AnkleLeft, PyKinectV2.JointType_SpineMid,
                PyKinectV2.JointType_Neck, PyKinectV2.JointType_SpineBase, PyKinectV2.JointType_HandRight,
                PyKinectV2.JointType_HandTipRight, PyKinectV2.JointType_ThumbRight, PyKinectV2.JointType_HandLeft,
                PyKinectV2.JointType_HandTipLeft, PyKinectV2.JointType_ThumbLeft, PyKinectV2.JointType_FootRight,
                PyKinectV2.JointType_FootLeft]
        left_most = float('inf')
        right_most = 0
        up_most = float('inf')
        down_most = 0
```

```python
        for No in range(25):
            pt = points[No]
            jointState = joints[pt].TrackingState;
            if jointState == PyKinectV2.TrackingState_NotTracked or jointState == PyKinectV2.TrackingState_Inferred:
                if self._is_get == True and No < 15:
                    X.append(None)
                    Y.append(None)
                continue
            x = jointPoints[pt].x
            y = jointPoints[pt].y
            if x < left_most:
                left_most = x
            if x > right_most:
                right_most = x
            if y < up_most:
                up_most = y
            if y > down_most:
                down_most = y
            if self._is_get == True and No < 15:
                # print(No)
                # print(pt)
                # print(x, y)
                if x < 0 or x > 1920:
                    X.append(None)
                else:
                    X.append(int(x * 600 / 1920))
                    # self._is_get = False
                    # self._interval = self._inter_range
                if y < 0 or y > 1080:
                    Y.append(None)
                else:
                    Y.append(int(y * 480 / 1080))
                    # self._is_get = False
                    # self._interval = self._inter_range


        if self._is_get:
            if len(X)==15 and len(Y)==15:
                self._x += X
                self._y += Y
            else:
                self._interval = self._inter_range
            self._is_get = False
            # print(len(self._x), len(self._y))
            # print(self._x)
            # print(self._y)
        height = down_most - up_most
        width = right_most - left_most

        corner1 = (left_most - 0.7*math.sqrt(abs(width)), up_most - 0.15*height)
        corner2 = (right_most + 0.7*math.sqrt(abs(width)), up_most - 0.15*height)
        corner3 = (right_most + 0.7*math.sqrt(abs(width)), down_most + 0.15*height)
        corner4 = (left_most - 0.7*math.sqrt(abs(width)), down_most + 0.15*height)
        try:
            pygame.draw.line(self._frame_surface, color, corner1, corner2, 8)
            pygame.draw.line(self._frame_surface, color, corner2, corner3, 8)
            pygame.draw.line(self._frame_surface, color, corner3, corner4, 8)
            pygame.draw.line(self._frame_surface, color, corner4, corner1, 8)

        except: # need to catch it due to possible invalid positions (with inf)
            pass

    def update_pose(self):
        if len(self._x) < 120 or len(self._y) < 120:
```

```
            return
        if len(self._x) > 120 or len(self._y) > 120:
            print("range error!")
            self._x = []
            self._y = []
            return
        data = np.array(self._x + self._y)
        data = np.reshape(data, (1, 240))
        self._x = []
        self._y = []

        data = data.reshape(-1,2,8,15)
        data = torch.Tensor(data)

        predict = np.argmax(self.net(data).data.numpy())
        self._pose = self.labels[predict]

        self._interval = 0


    def draw_color_frame(self, frame, target_surface):
        target_surface.lock()
        address = self._kinect.surface_as_array(target_surface.get_buffer())
        ctypes.memmove(address, frame.ctypes.data, frame.size)
        del address
        target_surface.unlock()

    def run(self):
        # -------- Main Program Loop -----------
        while not self._done:
            # --- Main event loop
            for event in pygame.event.get(): # User did something
                if event.type == pygame.QUIT: # If user clicked close
                    self._done = True # Flag that we are done so we exit this loop

                elif event.type == pygame.VIDEORESIZE: # window resized
                    self._screen = pygame.display.set_mode(event.dict['size'],
                                               pygame.HWSURFACE|pygame.DOUBLEBUF|pygame.RESIZABLE, 32)

            # --- Game logic should go here

            # --- Getting frames and drawing

            # --- Cool! We have a body frame, so can get skeletons
            if self._kinect.has_new_body_frame():
                self._bodies = self._kinect.get_last_body_frame()

            # --- draw skeletons to _frame_surface
            if self._bodies is not None:
                for i in range(0, self._kinect.max_body_count):
                    body = self._bodies.bodies[i]
                    if not body.is_tracked:
                        continue

                    joints = body.joints
                    # convert joint coordinates to color space
                    joint_points = self._kinect.body_joints_to_color_space(joints)
                    #self.draw_body(joints, joint_points, SKELETON_COLORS[i])


            # --- Woohoo! We've got a color frame! Let's fill out back buffer surface with frame's data
            if self._kinect.has_new_color_frame():
                self._interval += self._clock.get_time()
```

```
            if self._interval > self._inter_range:
                self._interval = 0
                self._is_get = True



            '''
            here is the main code to collect the dataset
            '''
            frame = self._kinect.get_last_color_frame()
            # reshape the frame into image shape
            image = frame.reshape((self._kinect.color_frame_desc.Height, self._kinect.color_frame_desc.width, 4),order='C')
            # get the rgb part and shrink them
            rgb_image = image[:, :, :3].copy()
            rgb_image = cv2.resize(rgb_image, (600, 480))
            # get the depth part and shrink it
            d_image = image[:, :, 3].copy()
            d_image = cv2.resize(d_image, (600, 480))
            d_image = np.reshape(d_image, (1,600,480))

            if self._is_get:
                if self._frame_num == 0:
                    os.mkdir(PATH+str(self._group_num)+'/')
                path = PATH + str(self._group_num) + '/'
                # save the rgb image
                cv2.imwrite(path+str(self._frame_num)+".jpg", rgb_image)
                # concatenate 16 depth data together
                self.depth = np.concatenate((self.depth, d_image), axis=0)
                self._frame_num += 1
                if self._frame_num == 16:
                    self.depth = np.delete(self.depth, 0, axis=0)
                    # save all the 16 frames' depth data
                    np.save(path+"depth.npy", self.depth)
                    self.depth = np.zeros((1,600,480))
                    self._frame_num = 0
                    self._group_num += 1



            # cv2.imshow('test', image)
            self.draw_color_frame(frame, self._frame_surface)
            frame = None



        # --- copy back buffer surface pixels to the screen, resize it if needed and keep aspect ratio
        # --- (screen size may be different from Kinect's color frame size)

        h_to_w = float(self._frame_surface.get_height()) / self._frame_surface.get_width()
        target_height = int(h_to_w * self._screen.get_width())
        surface_to_draw = pygame.transform.scale(self._frame_surface, (self._screen.get_width(), target_height))

        # # Display some text
        # font = pygame.font.Font(None, 60)
        # text = font.render(self._pose, 1, (10, 10, 10))
        # textpos = text.get_rect()
        # textpos.centerx = surface_to_draw.get_rect().centerx
        # surface_to_draw.blit(text, textpos)

        self._screen.blit(surface_to_draw, (0,0))
        pygame.display.update()
```

```
            # --- Go ahead and update the screen with what we've drawn.
            pygame.display.flip()

            # --- Limit to 60 frames per second
            self._clock.tick(60)

        # Close our Kinect sensor, close the window and quit.
        self._kinect.close()
        pygame.quit()


__main__ = "Kinect v2 Body Game"
game = BodyGameRuntime();
game.run();
```

5.4 Real-time Recognition

Python file `PyKinectBodyGame.py` is the demo of real-time action recognition system. It is a game based on `Pygame`. After running it, a game window will appear. Stand in front of it, you will see a box encircling you, which means you are sensed. After that, do some actions in the list [‘wave’, ‘drink’, ‘call’, ‘applause’, ‘stand’, ‘sit’, ‘stand still’], and the label it recognized will be shown on the screen.

```
from pykinect2 import PyKinectV2
from pykinect2.PyKinectV2 import *
from pykinect2 import PyKinectRuntime

from C2D_model import C2D
import torch
import torch.nn as nn
from torch.autograd import Variable

import ctypes
import _ctypes
import pygame
import sys
import math
import numpy as np
import cv2
import time
import torch

if sys.hexversion >= 0x03000000:
    import _thread as thread
else:
    import thread

# colors for drawing different bodies
SKELETON_COLORS = [pygame.color.THECOLORS["red"],
                   pygame.color.THECOLORS["blue"],
                   pygame.color.THECOLORS["green"],
                   pygame.color.THECOLORS["orange"],
                   pygame.color.THECOLORS["purple"],
                   pygame.color.THECOLORS["yellow"],
                   pygame.color.THECOLORS["violet"]]


class BodyGameRuntime(object):
    def __init__(self):
```

```python
        # initialize the game
        pygame.init()

        # Used to manage how fast the screen updates
        self._clock = pygame.time.Clock()

        # Set the width and height of the screen [width, height]
        self._infoObject = pygame.display.Info()
        self._screen = pygame.display.set_mode((self._infoObject.current_w >> 1, self._infoObject.current_h >> 1),
                                               pygame.HWSURFACE|pygame.DOUBLEBUF|pygame.RESIZABLE, 32)

        pygame.display.set_caption("Action Recognition with Kinect")

        # Loop until the user clicks the close button.
        self._done = False

        # Used to manage how fast the screen updates
        self._clock = pygame.time.Clock()

        # Kinect runtime object, we want only color and body frames
        self._kinect = PyKinectRuntime.PyKinectRuntime(PyKinectV2.FrameSourceTypes_Color | PyKinectV2.FrameSourceTypes_Body)

        # back buffer surface for getting Kinect color frames, 32bit color, width and height equal to the Kinect color frame size
        self._frame_surface = pygame.Surface((self._kinect.color_frame_desc.Width, self._kinect.color_frame_desc.Height), 0, 32)

        # here we will store skeleton data
        self._bodies = None

        # pose of user
        self._pose = "stand"

        #to get data
        self._is_get = False

        # data
        self._x = [] # x coordiates of joints points
        self._y = [] # y coordiates of joints points
        self.labels = ['wave', 'drink',"call",  "appaluse", "stand", "sit", "stand still"]

        # interval
        self._inter_range = 100
        self._interval = 0

        # build the net by loading pytorch model
        self.net = torch.load('net2.pkl')
        print("model loaded")

    # show a bone on the screen
    def draw_body_bone(self, joints, jointPoints, color, joint0, joint1):
        joint0State = joints[joint0].TrackingState;
        joint1State = joints[joint1].TrackingState;

        # both joints are not tracked
        if (joint0State == PyKinectV2.TrackingState_NotTracked) or (joint1State == PyKinectV2.TrackingState_NotTracked):
            return

        # both joints are not *really* tracked
        if (joint0State == PyKinectV2.TrackingState_Inferred) and (joint1State == PyKinectV2.TrackingState_Inferred):
```

```
            return

        # get the endpoints
        start = (jointPoints[joint0].x, jointPoints[joint0].y)
        end = (jointPoints[joint1].x, jointPoints[joint1].y)

        try:
            pygame.draw.line(self._frame_surface, color, start, end, 8)
        except: # need to catch it due to possible invalid positions (with inf)
            pass

    # draw the total body on the screen
    def draw_body(self, joints, jointPoints, color):
        # Torso
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_Head,
            PyKinectV2.JointType_Neck);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_Neck,
            PyKinectV2.JointType_SpineShoulder);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_SpineShoulder,
            PyKinectV2.JointType_SpineMid);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_SpineMid,
            PyKinectV2.JointType_SpineBase);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_SpineShoulder,
            PyKinectV2.JointType_ShoulderRight);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_SpineShoulder,
            PyKinectV2.JointType_ShoulderLeft);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_SpineBase,
            PyKinectV2.JointType_HipRight);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_SpineBase,
            PyKinectV2.JointType_HipLeft);

        # Right Arm
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_ShoulderRight,
            PyKinectV2.JointType_ElbowRight);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_ElbowRight,
            PyKinectV2.JointType_WristRight);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_WristRight,
            PyKinectV2.JointType_HandRight);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_HandRight,
            PyKinectV2.JointType_HandTipRight);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_WristRight,
            PyKinectV2.JointType_ThumbRight);

        # Left Arm
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_ShoulderLeft,
            PyKinectV2.JointType_ElbowLeft);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_ElbowLeft,
            PyKinectV2.JointType_WristLeft);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_WristLeft,
            PyKinectV2.JointType_HandLeft);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_HandLeft,
            PyKinectV2.JointType_HandTipLeft);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_WristLeft,
            PyKinectV2.JointType_ThumbLeft);

        # Right Leg
        #self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_HipRight,
            PyKinectV2.JointType_KneeRight);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_KneeRight,
            PyKinectV2.JointType_AnkleRight);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_AnkleRight,
            PyKinectV2.JointType_FootRight);

        # Left Leg
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_HipLeft,
```

```
            PyKinectV2.JointType_KneeLeft);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_KneeLeft,
            PyKinectV2.JointType_AnkleLeft);
        self.draw_body_bone(joints, jointPoints, color, PyKinectV2.JointType_AnkleLeft,
            PyKinectV2.JointType_FootLeft);


    # draw the box on the screen
    def draw_box(self, joints, jointPoints, color):
        '''
        function1: draw the box of body, according to the boundary of coordinates
        function2: store the data if it is time to sample
        '''
        X = []
        Y = []
        # calculate the interval from last sampling
        self._interval += self._clock.get_time()
        # if it is time to sample
        if self._interval > self._inter_range:
            self._interval = 0
            self._is_get = True
        # [3] [20] [8] [9] [10] [4] [5] [6] [16] [17] [18] [12] [13] [14] [1]
        # the first 15 joints are the data we need to input to the model
        # they have the same order with training datasets
        points = [PyKinectV2.JointType_Head, PyKinectV2.JointType_SpineShoulder, PyKinectV2.
            JointType_ShoulderRight,
                PyKinectV2.JointType_ElbowRight, PyKinectV2.JointType_WristRight, PyKinectV2
                    .JointType_ShoulderLeft,
                PyKinectV2.JointType_ElbowLeft, PyKinectV2.JointType_WristLeft, PyKinectV2.
                    JointType_HipRight,
                PyKinectV2.JointType_KneeRight, PyKinectV2.JointType_AnkleRight, PyKinectV2.
                    JointType_HipLeft,
                PyKinectV2.JointType_KneeLeft, PyKinectV2.JointType_AnkleLeft, PyKinectV2.
                    JointType_SpineMid,
                PyKinectV2.JointType_Neck, PyKinectV2.JointType_SpineBase, PyKinectV2.
                    JointType_HandRight,
                PyKinectV2.JointType_HandTipRight, PyKinectV2.JointType_ThumbRight,
                    PyKinectV2.JointType_HandLeft,
                PyKinectV2.JointType_HandTipLeft, PyKinectV2.JointType_ThumbLeft, PyKinectV2
                    .JointType_FootRight,
                PyKinectV2.JointType_FootLeft]
        # initialize the boundary
        left_most = float('inf')
        right_most = 0
        up_most = float('inf')
        down_most = 0
        # traverse all the joints
        for No in range(25):
            pt = points[No]
            jointState = joints[pt].TrackingState
            # if the joint is not tracked(missing) and we need to get it
            # store the coordinates as None and continue
            if jointState == PyKinectV2.TrackingState_NotTracked or jointState == PyKinectV2
                .TrackingState_Inferred:
                if self._is_get == True and No < 15:
                    X.append(None)
                    Y.append(None)
                continue
            # update the boundary
            x = jointPoints[pt].x
            y = jointPoints[pt].y
            if x < left_most:
                left_most = x
            if x > right_most:
                right_most = x
```

```
            if y < up_most:
                up_most = y
            if y > down_most:
                down_most = y
            # if it is time to sample
            if self._is_get == True and No < 15:
                # if out of range, append None
                if x < 0 or x > 1920:
                    X.append(None)
                else:
                    # change the size of coordinates and store
                    X.append(600 - int(x * 600 / 1920))
                if y < 0 or y > 1080:
                    Y.append(None)
                else:
                    Y.append(int(y * 480 / 1080))

        # append the data of this frame to the total list
        if self._is_get:
            if len(X)==15 and len(Y)==15:
                self._x += X
                self._y += Y
            # if data is not complete, sample next frame
            # but I think it is not gonna happen
            else:
                self._interval = self._inter_range
        height = down_most - up_most
        width = right_most - left_most

        # since the joints is not the strict boundary
        # change the coordinates
        corner1 = (left_most - 0.7*math.sqrt(abs(width)), up_most - 0.15*height)
        corner2 = (right_most + 0.7*math.sqrt(abs(width)), up_most - 0.15*height)
        corner3 = (right_most + 0.7*math.sqrt(abs(width)), down_most + 0.15*height)
        corner4 = (left_most - 0.7*math.sqrt(abs(width)), down_most + 0.15*height)
        # draw the box
        try:
            pygame.draw.line(self._frame_surface, color, corner1, corner2, 8)
            pygame.draw.line(self._frame_surface, color, corner2, corner3, 8)
            pygame.draw.line(self._frame_surface, color, corner3, corner4, 8)
            pygame.draw.line(self._frame_surface, color, corner4, corner1, 8)

        except: # need to catch it due to possible invalid positions (with inf)
            pass

    # filter the None data
    '''
    if we get None in draw_box()
    set them with approximate values
    '''
    def filter_none_data(self, features):
        for i in range(len(features)):
            # set None to 0
            non_pos = []
            for j in range(len(features[i])):
                if str(features[i][j]) == str(None):
                    non_pos.append(j)
                    features[i][j] = 0
            # if it is 0, set it with average of neighbers
            for k in range(10): #value iteration times
                for m in range(len(non_pos)):
                    if non_pos[m]%15 == 0:  #begin position
                        features[i][non_pos[m]] = features[i][non_pos[m]+1]
                    elif (non_pos[m]+1) % 15 == 0: #end position
                        features[i][non_pos[m]] = features[i][non_pos[m]-1]
```

```python
                    else:#medium position
                        features[i][non_pos[m]] = (features[i][non_pos[m]+1] + features[i][non_pos[m]-1]) / 2
        return features


    def update_pose(self):
        '''
        update the pose by feed the data into the model
        show the label in the screen
        '''
        # if data is not enough, skip
        if len(self._x) < 240 or len(self._y) < 240:
            return
        # if data out of range, raise error
        if len(self._x) > 240 or len(self._y) > 240:
            raise ValueError("out of range")
        Time = time.time()
        data = np.array(self._x + self._y)
        data = np.reshape(data, (1, 480))
        data = self.filter_none_data(data)
        self._x = []
        self._y = []

        data = data.astype(np.int32)
        # reshape the data to fit the model
        data = data.reshape(-1,2,16,15)
        # turn into torch tensor
        data = torch.Tensor(data)

        predict = np.argmax(self.net(data).data.numpy())
        # update the label
        self._pose = self.labels[predict]
        # restart the interval
        self._interval = 0
        # show the time of predict
        print(time.time()-Time)


    # draw the frame on the screen
    def draw_color_frame(self, frame, target_surface):
        target_surface.lock()
        address = self._kinect.surface_as_array(target_surface.get_buffer())
        ctypes.memmove(address, frame.ctypes.data, frame.size)
        del address
        target_surface.unlock()

    # run pygame
    def run(self):
        # -------- Main Program Loop -----------
        while not self._done:
            # Main event loop
            for event in pygame.event.get(): # User did something
                if event.type == pygame.QUIT: # If user clicked close
                    self._done = True # Flag that we are done so we exit this loop

                elif event.type == pygame.VIDEORESIZE: # window resized
                    self._screen = pygame.display.set_mode(event.dict['size'],
                                            pygame.HWSURFACE|pygame.DOUBLEBUF|pygame.RESIZABLE, 32)


            # Getting frames and drawing
            if self._kinect.has_new_color_frame():
                frame = self._kinect.get_last_color_frame()
```

```python

                # it can be reshape into image matrix(RGB-D)
                # image = frame.reshape((self._kinect.color_frame_desc.Height, self._kinect.
                    color_frame_desc.width, 4),order='C')
                # image = image[:, :, :3]
                # cv2.imshow('test', image)

                self.draw_color_frame(frame, self._frame_surface)
                frame = None

            # We have a body frame, so can get skeletons
            if self._kinect.has_new_body_frame():
                self._bodies = self._kinect.get_last_body_frame()

            # draw skeletons to _frame_surface
            if self._bodies is not None:
                # traverse the bodies
                for i in range(0, self._kinect.max_body_count):
                    body = self._bodies.bodies[i]
                    if not body.is_tracked:
                        continue

                    joints = body.joints
                    # convert joint coordinates to color space
                    joint_points = self._kinect.body_joints_to_color_space(joints)
                    # draw the box and store the data
                    self.draw_box(joints, joint_points, SKELETON_COLORS[i])

            # copy back buffer surface pixels to the screen, resize it if needed and keep
                aspect ratio
            # (screen size may be different from Kinect's color frame size)
            self.update_pose()
            h_to_w = float(self._frame_surface.get_height()) / self._frame_surface.get_width
                ()
            target_height = int(h_to_w * self._screen.get_width())
            surface_to_draw = pygame.transform.scale(self._frame_surface, (self._screen.
                get_width(), target_height))

            # Display the action label
            font = pygame.font.Font(None, 60)
            text = font.render(self._pose, 1, (255, 10, 10))
            textpos = text.get_rect()
            textpos.centerx = surface_to_draw.get_rect().centerx
            surface_to_draw.blit(text, textpos)
            self._screen.blit(surface_to_draw, (0,0))
            # update the frame
            pygame.display.update()

            # Go ahead and update the screen with what we've drawn.
            pygame.display.flip()

            # Limit the fps
            self._clock.tick(20)

        # Close our Kinect sensor, close the window and quit.
        self._kinect.close()
        pygame.quit()


__main__ = "Kinect v2 Body Game"
game = BodyGameRuntime();
game.run();
```

6 Discussion and Conclusions

6.1 Discussion

6.1.1 Missing Data Processing

Openpose is based on RGB graph. In some condition it's not as powerful as we thought. For instance, if one's arm is hidden behind his/her body, Openpose cannot recognize, or very inaccurate. We thought of two solutions for this problem. First is to infer the missing point with its neighbors. Second is to implement depth graph. For the first solution, we just use 10 times value iteration to fill these values by neighboring ones. However, it's not so sensible because the relationship of these values is not evidently known by us. When we want to add boxing and walking to our actions, we found the net performs badly due to the roughly processing of None values. The second solution will be introduced later.

In the future, we may try to use HMM to help us with padding these values.

6.1.2 The Use of Depth Information

The Kinect can provide us with RGB-D images and skeleton structure. First, we use RGB images to train our 3D convolution model. Due to the large number of dimensions and many irrelevant and useless information, our first trying only attended nearly 25% accuracy. Then, we use the key points of the skeleton structure in our net mentioned above and obtained not bad results.

However, the remaining depth images we didn't use is also a key breakthrough point. We saved the depth images when we created our own dataset. At that time, because of running out of memory when we load the depth data, we didn't use it. Then we reduced the dimension of the depth images and changed the structure of our model to use both key points and depth information. What finally frustrated us was that when we were ready to load the small size depth data, we found that the depth data were all 255! However, we don't have enough time to capture the actions again. What a shame...

6.1.3 Our New Dataset

As for our new dataset. There are three aspects we want to improve.

First, we want to add high-quality depth images, shown in Figure 12. Although we collected depth images in the first vision, we found that the matrix contains 255 mostly, which means it is not useful while training. We believe it is because we stood too close to the wall at that time so that the Kinect could not distinguish the depth precisely. We plan to record the depth images one more time during which we will stand far from the background. But here comes another question, with different distance from the background we will get totally different data range in depth images. What if we have new testing situations with various distances from the background? Can our model learn the generalization ability training with monotonous background dataset? That's the thing we have to figure out if we use depth images to train our model.

Figure 12: We intend to implement depth graphs.

Second, we plan to expand our action pool with some interactive actions, for example, fighting and embracing (Figure 13). They are also very useful in real life application. It is also very challenging. How does our model know that it is time to recognize interactive actions rather than single actions of 2 people? What's more, in interactive actions, parts of players' bodies will sure to overlap. It is hard to get joints coordinates with such an incomplete information and it is also very hard to predict actions with them.

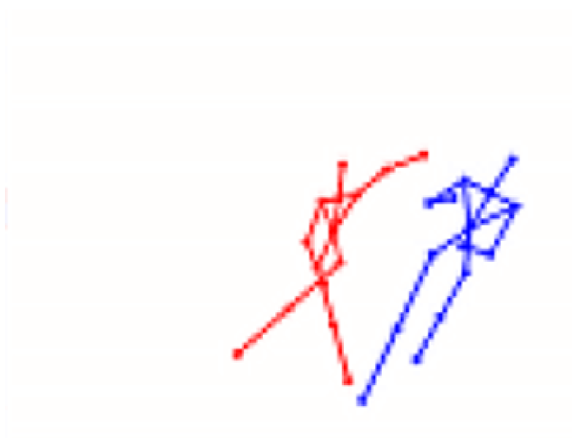

Figure 13: We intend to add some interactive actions.

Finally, we plan to expand the dataset with data containing more people. Limited by time and resources, we only collected data of the group members. Of course, it is not big enough because different people have different bodily form and action habits. Those are import part of the model generalization ability.

6.1.4 Real-time Video Sampling

We found that it still makes some mistakes during interval of actions. It is because the data during the interval of different actions is not related to any action. Of course, we did not train our model with this kind of data. It is hard to predict actions like that because they are not defined actions at all. We plan to handle it like this. We only show the action label when it is confirmed by our model. We can set the threshold value as 50%, only when the possibility is more than it can we confirm the prediction.

6.2 Conclusion

In this project, we presented a convolution network to perform human action recognition. We used OpenPose to extract joints coordinates which can show key information of a person's posture. We concatenated series of coordinates and input them to get prediction label. This CNN is first trained on FLORENCE 3D ACTIONS DATASET, which is very small with 8 frames formed as a sample. To improve the generalization ability of the CNN model we collected our own dataset called Action-209. We designed it according to the shortcomings of the previous one. Then we adjusted the structure CNN to fit the sample size of 16 frames and get testing accuracy at nearly 95%. The model is also applied on a real-time action recognition system. It can do multi-targets recognition at the same time and perform with considerable accuracy. That means our model does have practical application value.

References

[1] Diogo C. Luvizon1, David Picard, Hedi Tabia. “2D/3D Pose Estimation and Action Recognition using Multitask Deep Learning,” in CVPR, 2018

[2] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri. “Learning spatiotemporal features with 3d convolutional networks,” in ICCV, 2015.

[3] S. Ji,W. Xu, M. Yang, and K. Yu. “3d convolutional neural networks for human action recognition,” IEEE Trans. Pattern Analysis and Machine Intelligence, 2013.

[4] C. Feichtenhofer, A. Pinz, and R. P. Wildes. “Spatiotemporal multiplier networks for video action recognition,” in 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2017, pp. 7445–7454.

[5] K. Simonyan and A. Zisserman. “two-stream convolutional networks for action recognition in videos,” in NIPS, 2014.

[6] G. Varol, I. Laptev, and C. Schmid. “Long-term temporal convolutions for action recognition,” IEEE Transactions on Pattern Analysis and Machine Intelligence, 2017.

[7] Zhe Cao and Gines Hidalgo and Tomas Simon and Shih-En Wei and Yaser Sheikh. “OpenPose: realtime multi-person 2D pose estimation using Part Affinity Fields,” arXiv preprint arXiv:1812.08008, 2018.

[8] Zhe Cao and Tomas Simon and Shih-En Wei and Yaser Sheikh. “Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields,” CVPR, 2017.

[9] Tomas Simon and Hanbyul Joo and Iain Matthews and Yaser Sheikh. “Hand Keypoint Detection in Single Images using Multiview Bootstrapping,” CVPR, 2017.

[10] Shih-En Wei and Varun Ramakrishna and Takeo Kanade and Yaser Sheikh. “Convolutional pose machines,” CVPR, 2016.