Jump Detection Project Report

Final Submission

April 2020

Shaohua Yuan

UIN: 629009213

Email: shyuan@tamu.edu

New Contribution

1. Dataset

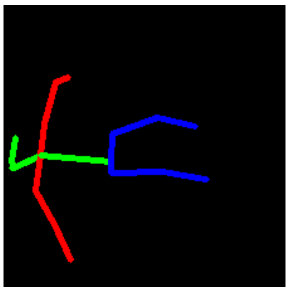
a) Expand dataset

In the previous submission, we divided our dataset to two balanced parts. The positive dataset contains videos of jumping. The negative dataset contains videos of running, walking, sitting down, exercising and standing up. However, in the real application, I found that the model had confusion when the person in the video just stands still and does not move. Therefore, to solve this problem, I include the action of standing still in the negative dataset. To balance the dataset, I also add more videos of jumping to the positive dataset.

All new videos added to the dataset are the videos I shoot myself jumping or standing still. For positive dataset, I added 209 3-seconds clips of jumping into it. For negative dataset, I added 375 3-seconds clips of standing still into it. Finally, we have 2180 videos for training, 230 videos for validating, 232 videos for testing. The videos are from Weizmann, UCF-101, STAIR-actions, and I shoot myself.

b) Data augmentation

In this project, I use OpenCV to capture videos and use the neural-network model to analyze it. In the previous submissions, I assumed that the videos captured by OpenCV is in the correct direction, but the truth is the captured video may be rotated 90-degree, 180-degree or 270-degree, which depends on the camera. The same happens to the body landmarks. Like the follows:



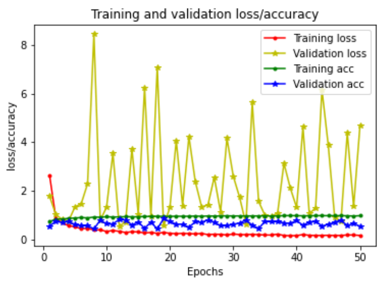
To solve the problems, the neural-network model needs to be able to analyze videos from different rotation degrees. Therefore, I augment the dataset by rotating the videos and landmarks by 90-degree, 180-degree and 270-degree. Therefore, after data augmentation, the size of dataset is expanded fourfold. Finally, we have 8720 videos for training, 920 videos for validating, 928 videos for testing.

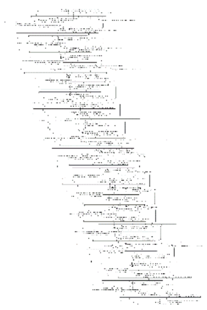
2. New ideas and models

a) 3D ResNet model

In previous submission, my model totally relies on the body landmarks as input. To further improve the performance, the information from RGB frames is also useful. In order to analyze RGB frames, I choose to adopt 3D ResNet model.

The model is from JihongJu, its structure and performance is as follows:





(click and see detail in browser)

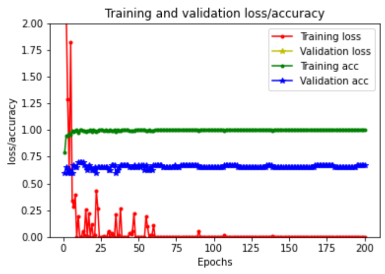
Its best performance on the test dataset is test_acc:0.7241379022598267, test_loss:0.7444219887256622. Although its best performance is better than base case 50%, from the train/valid figure we can tell that the model simply remembers the train dataset and fails to extract useful features that can be applied to validation dataset. Therefore, the 3D ResNet model will not work in detecting jumping.

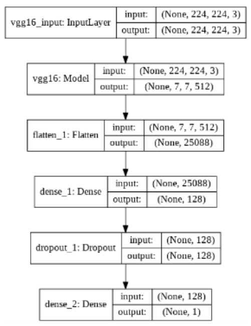
b) 2D CNN model

Since 3D ResNet model fails, we need to figure out whether we can get useful information from RGB frames in order to detect jumping. The most efficient way is to use 2D CNN model to classify those frames. The frames can be divided to two classes. One is positive class, the person in the positive frames is jumping, i.e., the person's body is on the air and the feet is not touching the ground. One is negative class, the persons in the negative frames is doing something with his feet touching the ground. As follows, the left is positive, right is negative:

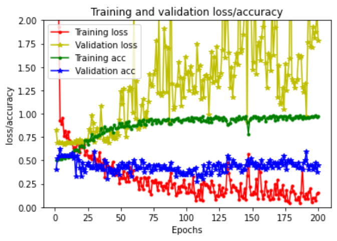
Then I build a small dataset contains these RGB frames. It has 218 images for training, 40 images for validating, 43 images for testing.

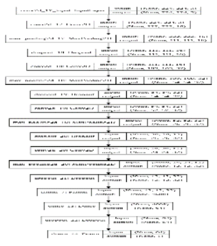
I first use transfer learning to use VGG16 with ImageNet weights, its structure and performance are:





Clearly, it is overfitting. The size of our dataset is small, and to get a better performance we need to reduce the size of our CNN model. The structure and performance of the small CNN model are:





(click and see detail in browser)

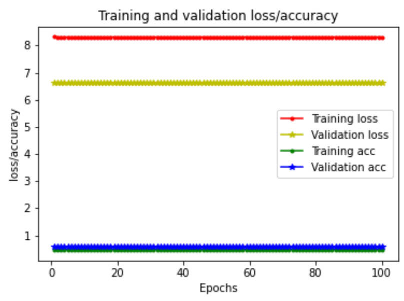
Its best performance on the test dataset is test_acc:0.5348837375640869, test_loss:0.6902745047280955. From its performance and its train/valid figure, we conclude that 2D CNN model cannot classify jumping/not jumping images very well. This infers that RGB frames are not very useful in detecting jumps. Therefore, there is no reason for us to include RGB frames in our model.

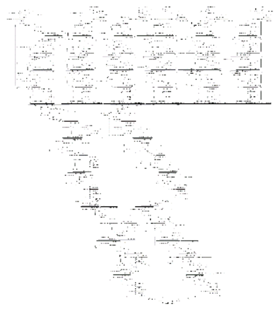
c) Pretrained SSD model

I read the blog regarding the SSD model. However, this model is not very useful in improving the performance, because it only detects the border of the person, while OpenPose detects the exact skeleton location of the person. Since we have already included OpenPose in our project, there is no need to include SSD model.

d) Temporal CNN model

Because we have concluded that RGB frames is not useful in detecting jumping, in order to further improve the performance, we need to improve the model which totally relies on the body landmarks as its input. After reading the paper, I notice that temporal CNN can be used to the analyze temporal information among frames, i.e. the temporal CNN can replace the role of RNN. Then I implemented the model, the Temporal Convolutional layer comes from philipperemy. The model's structure and performance are:





(click and see detail in browser)

Its best performance on the test dataset is test_acc:0.5387930870056152, test_loss:7.114092629531334. Obviously, the model does not work for the detecting jump purpose.

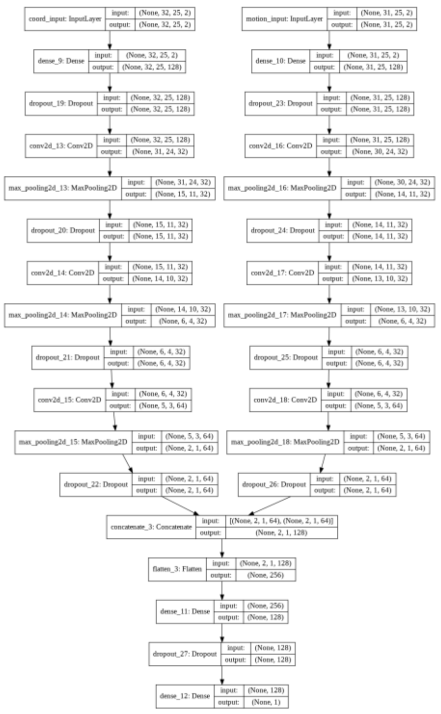
e) **(Important)** Ensemble learning

We have tried lots of new models, but none of them works well for the purpose of detecting jump. One technique that can be used to improve the performance without creating new model is called ensemble learning.

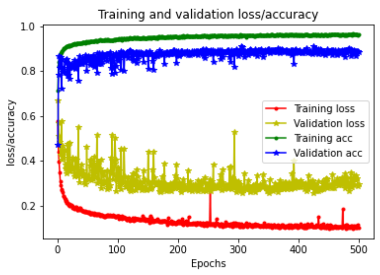
Actually, in submission2 and submission3, we have already got two good models that work very well on detecting jumping. One of them, the model based on RNN, have accuracy 93%, another, the model based on CNN, has accuracy 95%. We can ensemble the two model to create a new model, the new model should have a better performance than both of them.

Because we have expanded our dataset and used data augmentation. We need first train the two models on the new dataset again.

For the model based on CNN, which is based on *Skeleton-based Action Recognition with Convolutional Neural Networks*, since the model has very small number of parameters, we decide to train a brand-new model on the new dataset. Its structure and performance are:

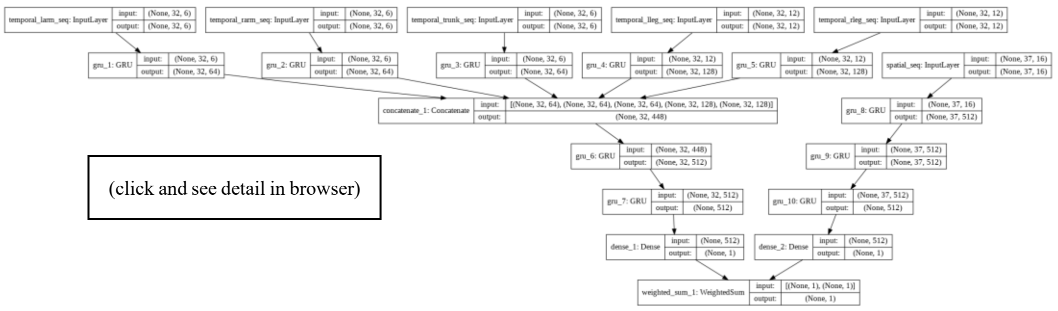


(click and see detail in browser)

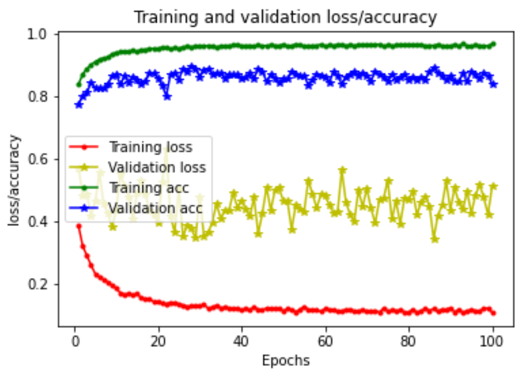


Its best performance on the new test dataset is test_acc:0.951508641242981, test_loss:0.12914977042839446.

For the model based on RNN, which is based on Modeling Temporal Dynamics and Spatial Configurations of Actions Using Two-Stream Recurrent Neural Networks, since the model has a large number of parameters, it takes time to train a brand-new model on the dataset. We decide to continue train the model we got in submission2 on the new dataset and get a new RNN based model. Its structure and performance are:

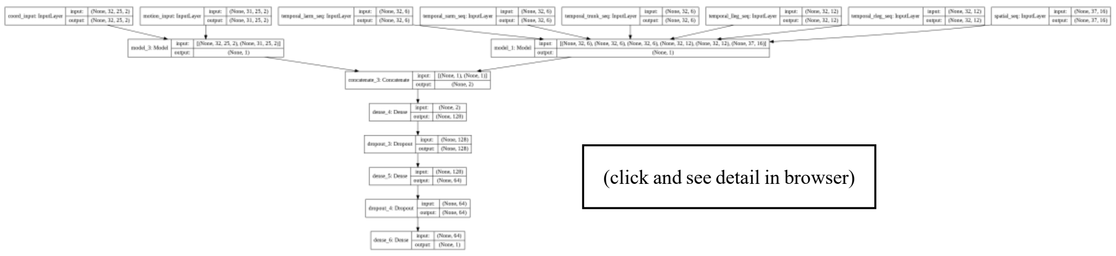


(click and see detail in browser)

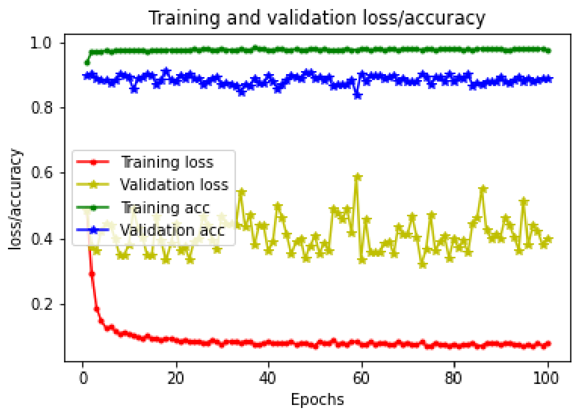


Its best performance on the new test dataset is test_acc:0.9482758641242981, test_loss:0.17175677033333941.

Then it is time to ensemble the two model, the ensemble method we used is stacking. We use one layer of Concatenate and three layers of Fully-Connected network, to stack the two sub-models. The new model's structure and performance are:



(click and see detail in browser)



Its best performance on the new test dataset is test_acc:0.96875, test_loss:0.1035012679881063.

Model	Test acc	Test loss
CNN based	95%	0.13
RNN based	95%	0.17
Ensemble	97%	0.10

From the above tables, we can conclude that ensemble learning really helps to improve the model. Finally, the ensemble model becomes the final model of our project. For more details of this model, like its limitations and advantages, please see the performance section in this report.

Report

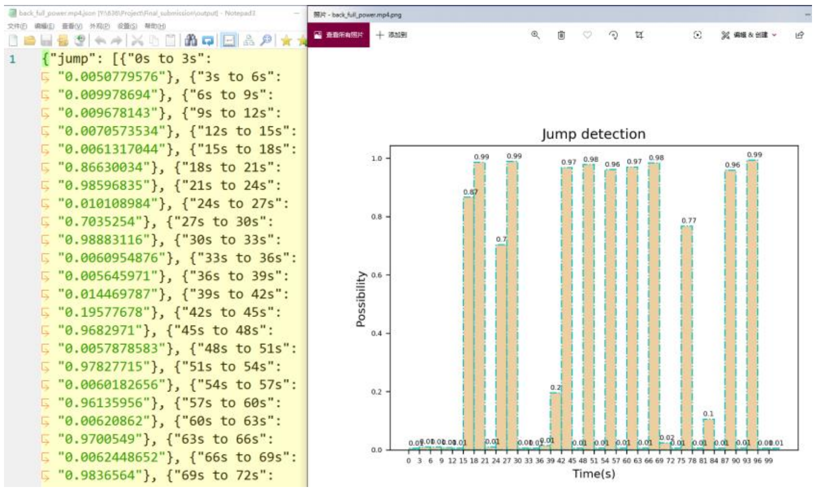
1. Topic

a) Introduction:

I will build a neural network project that can detect whether and when the human action of jump presents in videos. This project can be used in smart home applications for detecting human action of jump.

b) Solution:

- For a video, cut it into small clips whose time-length are fixed to 3 seconds.
- Use the OpenPose project to generate landmark json files for each clip.
- Let one clip's landmark files be the input tensor of the neural network, the output will be the possibility that jump is detected in this clip.
- For a video, input all its clips to the neural network in time order, and get the outputs in time order.
- Collect the outputs and generate a figure and json file, statistically showing jump detection in the video. As following:



```
{"jump": [{"0s to 3s":
 "0.0050779576"}, {"3s to 6s":
 "0.009978694"}, {"6s to 9s":
 "0.009678143"}, {"9s to 12s":
 "0.0070573534"}, {"12s to 15s":
 "0.0061317044"}, {"15s to 18s":
 "0.86630034"}, {"18s to 21s":
 "0.98596835"}, {"21s to 24s":
 "0.010108984"}, {"24s to 27s":
 "0.7035254"}, {"27s to 30s":
 "0.98883116"}, {"30s to 33s":
 "0.0060954876"}, {"33s to 36s":
 "0.005645971"}, {"36s to 39s":
 "0.014469787"}, {"39s to 42s":
 "0.19577678"}, {"42s to 45s":
 "0.9682971"}, {"45s to 48s":
 "0.0057878583"}, {"48s to 51s":
 "0.97827715"}, {"51s to 54s":
 "0.0060182656"}, {"54s to 57s":
 "0.96135956"}, {"57s to 60s":
 "0.00620862"}, {"60s to 63s":
 "0.9700549"}, {"63s to 66s":
 "0.0062448652"}, {"66s to 69s":
 "0.9836564"}, {"69s to 72s":
```

2. Dataset

a) Raw dataset: video

I use videos from Weizmann, UCF-101, STAIR-actions, and I also included the videos that I shoot myself. These are all datasets including different human actions (jump, run, walk, sit down, exercise, etc.). I collect around 1000 videos from these datasets. Half of their topics are jump, another half videos' topics are other actions (run, walk, sit down, exercise, stand up, stand still).

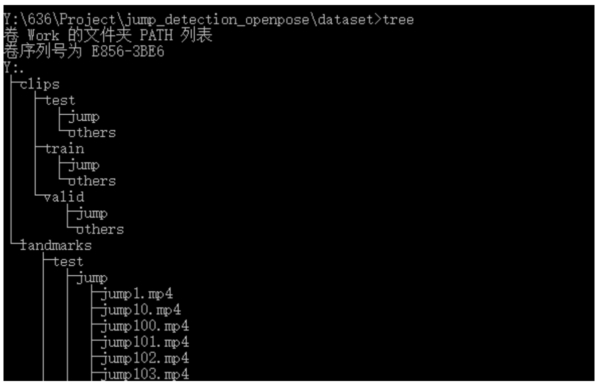
b) Process data: 3-seconds clips and body-landmarks:

- Clips: Use ffmpeg and Python script to split all videos into 3-seconds clips
- Landmarks: Use OpenPose alongside with Google Colab & Google Drive to generate landmark json files for each clip. Use the following command in Colab:

```
!cd openpose && ./build/examples/openpose/openpose.bin --video 'root_of_video' --write_json
'directory_of_output' --display 0 --render_pose 0 -keypoint_scale 3
```

```
!cd openpose && ./build/examples/openpose/openpose.bin --video sample/bed.mp4 --write_json sample/bed/ --display 0 --render_pose 0 -keypoint_scale 3
```

c) Dataset Directory Structure:



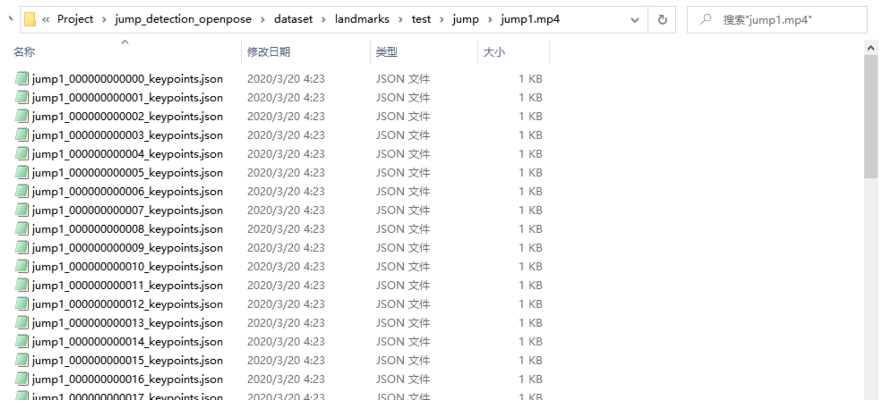
```
Y:\636\Project\jump_detection_openpose\dataset>tree
卷 Work 的文件夹 PATH 列表
卷序列号为 E856-3BE6
Y:.
├─clips
│  ├─test
│  │  ├─jump
│  │  └─others
│  ├─train
│  │  ├─jump
│  │  └─others
│  └─valid
│      ├─jump
│      └─others
└─landmarks
    ├─test
    │  ├─jump
    │  │  ├─jump1.mp4
    │  │  ├─jump10.mp4
    │  │  ├─jump100.mp4
    │  │  ├─jump101.mp4
    │  │  ├─jump102.mp4
    │  │  ├─jump103.mp4
```

Since there are lots of sub-directories under the “dataset/landmarks/” directory, we only post a part of command “tree” output.

The whole dataset (clips and body-landmark files) is stored in “dataset/” directory. Clips are stored in “dataset/clips/”, while landmark files are stored in “dataset/landmarks/”.

We get 1000 videos from Weizmann, UCF-101, STAIR-actions, and the videos I shoot myself. These videos are split into around 2700 clips. We choose 2300 of them as train clips, 200 of them as valid clips, and the lefts 200 clips as test clips.

For each clip, there exist a directory under “dataset/landmarks/” that has the same name as the clip name. For example, for video “dataset/clips/test/jump/jump1.mp4”, there is a directory “dataset/landmarks/test/jump/jump1.mp4/”, and this directory stores the landmark json files of video “dataset/clips/test/jump/jump1.mp4”, as following:



« Project › jump_detection_openpose › dataset › landmarks › test › jump › jump1.mp4

搜索"jump1.mp4"

名称	修改日期	类型	大小
jump1_000000000000_keypoints.json	2020/3/20 4:23	JSON 文件	1 KB
jump1_000000000001_keypoints.json	2020/3/20 4:23	JSON 文件	1 KB
jump1_000000000002_keypoints.json	2020/3/20 4:23	JSON 文件	1 KB
jump1_000000000003_keypoints.json	2020/3/20 4:23	JSON 文件	1 KB
jump1_000000000004_keypoints.json	2020/3/20 4:23	JSON 文件	1 KB
jump1_000000000005_keypoints.json	2020/3/20 4:23	JSON 文件	1 KB
jump1_000000000006_keypoints.json	2020/3/20 4:23	JSON 文件	1 KB
jump1_000000000007_keypoints.json	2020/3/20 4:23	JSON 文件	1 KB
jump1_000000000008_keypoints.json	2020/3/20 4:23	JSON 文件	1 KB
jump1_000000000009_keypoints.json	2020/3/20 4:23	JSON 文件	1 KB
jump1_000000000010_keypoints.json	2020/3/20 4:23	JSON 文件	1 KB
jump1_000000000011_keypoints.json	2020/3/20 4:23	JSON 文件	1 KB
jump1_000000000012_keypoints.json	2020/3/20 4:23	JSON 文件	1 KB
jump1_000000000013_keypoints.json	2020/3/20 4:23	JSON 文件	1 KB
jump1_000000000014_keypoints.json	2020/3/20 4:23	JSON 文件	1 KB
jump1_000000000015_keypoints.json	2020/3/20 4:23	JSON 文件	1 KB
jump1_000000000016_keypoints.json	2020/3/20 4:23	JSON 文件	1 KB
jump1_000000000017_keypoints.json	2020/3/20 4:23	JSON 文件	1 KB

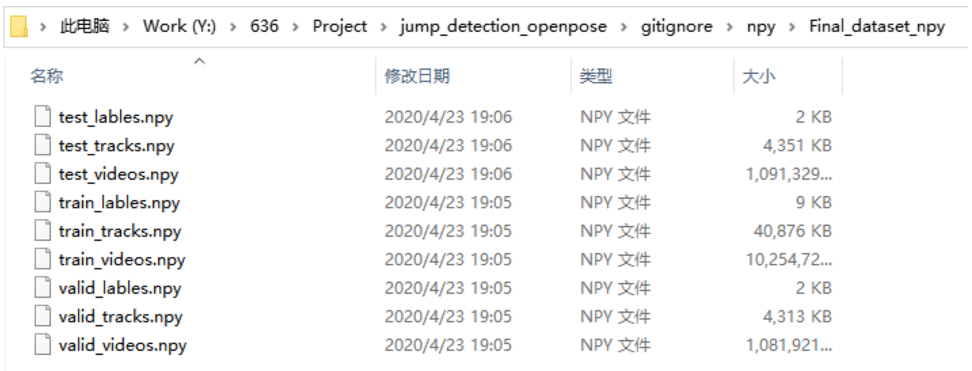
The number of json files equals to the number of frames in that clip.

d) Load dataset:

There are two ways of loading dataset:

The first way is to use OpenCV to load clips, and json python library to load landmark files. However, this is very slow, because we have so many files to read and I/O operation is much slower than CPU, so we have the second way:

After we use the first way to load all dataset, we use numpy to store this data as .npy files. Therefore, when we want to train our model again, we only need to load these .npy files, which is much faster. The files and code are as follows:

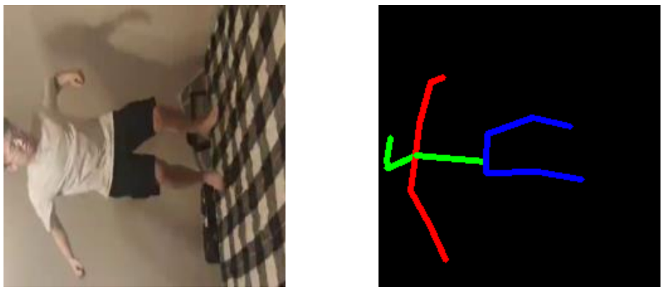

此电脑 › Work (Y:) › 636 › Project › jump_detection_openpose › gitignore › npy › Final_dataset_npy

名称	修改日期	类型	大小
test_lables.npy	2020/4/23 19:06	NPY 文件	2 KB
test_tracks.npy	2020/4/23 19:06	NPY 文件	4,351 KB
test_videos.npy	2020/4/23 19:06	NPY 文件	1,091,329...
train_lables.npy	2020/4/23 19:05	NPY 文件	9 KB
train_tracks.npy	2020/4/23 19:05	NPY 文件	40,876 KB
train_videos.npy	2020/4/23 19:05	NPY 文件	10,254,72...
valid_lables.npy	2020/4/23 19:05	NPY 文件	2 KB
valid_tracks.npy	2020/4/23 19:05	NPY 文件	4,313 KB
valid_videos.npy	2020/4/23 19:05	NPY 文件	1,081,921...

```
import numpy as np
train_videos = np.load("train_videos.npy")
train_tracks = np.load("train_tracks.npy")
train_lables = np.load("train_lables.npy")
valid_videos = np.load("valid_videos.npy")
valid_tracks = np.load("valid_tracks.npy")
valid_lables = np.load("valid_lables.npy")
test_videos = np.load("test_videos.npy")
test_tracks = np.load("test_tracks.npy")
test_lables = np.load("test_lables.npy")
```

e) Data Augmentation

When using OpenCV to capture videos, it is possible that the captured video will be rotated 90-degree, 180-degree or 270-degree, which depends on the camera. The same happens to the body landmarks. Like the follows:



To solve the problems, the neural-network model needs to be able to analyze videos from different rotation degrees. Therefore, I augment the dataset by rotating the videos and landmarks by 90-degree, 180-degree and 270-degree. Therefore, after data augmentation, the size of dataset is expanded fourfold. Finally, we have 8720 videos for training, 920 videos for validating, 928 videos for testing.

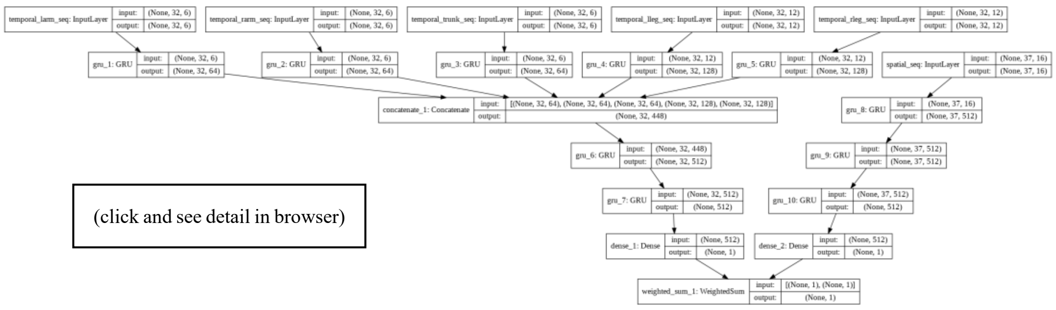
3. Network Model

a) Overview:

In this submission, the new model comes from the ensemble of two models from submission2 and submission3. Let's first get review on the two modes.

b) The model from submission2

For submission2, the model is an RNN based model. It is implemented based on Modeling Temporal Dynamics and Spatial Configurations of Actions Using Two-Stream Recurrent Neural Networks. It has the following structure:


temporal_larm_seq: InputLayer | input: (None, 32, 6) | output: (None, 32, 6)
temporal_rarm_seq: InputLayer | input: (None, 32, 6) | output: (None, 32, 6)
temporal_trunk_seq: InputLayer | input: (None, 32, 6) | output: (None, 32, 6)
temporal_lleg_seq: InputLayer | input: (None, 32, 12) | output: (None, 32, 12)
temporal_rleg_seq: InputLayer | input: (None, 32, 12) | output: (None, 32, 12)
gru_1: GRU | input: (None, 32, 6) | output: (None, 32, 64)
gru_2: GRU | input: (None, 32, 6) | output: (None, 32, 64)
gru_3: GRU | input: (None, 32, 6) | output: (None, 32, 64)
gru_4: GRU | input: (None, 32, 12) | output: (None, 32, 128)
gru_5: GRU | input: (None, 32, 12) | output: (None, 32, 128)
spatial_seq: InputLayer | input: (None, 37, 16) | output: (None, 37, 16)
concatenate_1: Concatenate | input: [(None, 32, 64), (None, 32, 64), (None, 32, 64), (None, 32, 128), (None, 32, 128)] | output: (None, 32, 448)
gru_8: GRU | input: (None, 37, 16) | output: (None, 37, 512)
gru_6: GRU | input: (None, 32, 448) | output: (None, 32, 512)
gru_9: GRU | input: (None, 37, 512) | output: (None, 37, 512)
gru_7: GRU | input: (None, 32, 512) | output: (None, 512)
gru_10: GRU | input: (None, 37, 512) | output: (None, 512)
dense_1: Dense | input: (None, 512) | output: (None, 1)
dense_2: Dense | input: (None, 512) | output: (None, 1)
weighted_sum_1: WeightedSum | input: [(None, 1), (None, 1)] | output: (None, 1)

(click and see detail in browser)

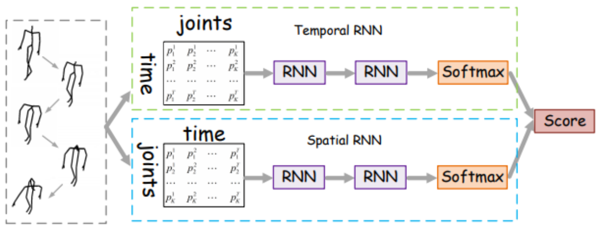


Figure 1. A two-stream RNN architecture for skeleton based action recognition. Here *Softmax* denotes a fully connected layer with a softmax activation function.

This model takes body-landmarks to two-stream, temporal stream and spatial stream, and do parallel computing, and finally collect the outputs from two stream and compute the final result.

<u>Temporal stream:</u>

This temporal channel of RNN models the temporal dynamics of skeletons. Here we use Hierarchical RNN, the definition of Hierarchical RNN is:

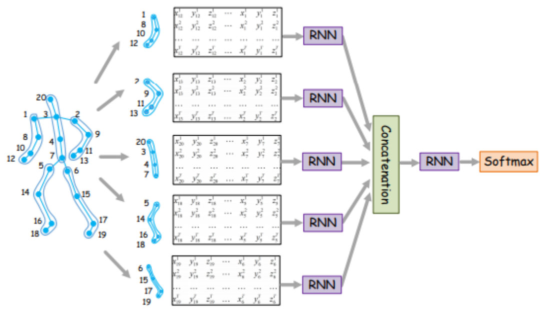


Figure 3. Hierarchical RNN for skeleton based action recognition.

The human skeleton can be divided into five parts, i.e., two arms, two legs and one trunk. We observe that an action is performed by either an independent part or a combination of several parts. For example, kicking depends on legs and running involves both legs and arms. Thus, a hierarchical structure of RNN is used to model the motions of different parts as well as the whole body. Figure 3 shows the proposed structure (From the paper).

The temporal stream is designed based on Hierarchical RNN, which takes body-landmarks of left arm, right arm, trunk, left leg and right leg separately as inputs, and follows three layers RNN, and a final dense layer with sigmoid activation function.

<u>Spatial stream:</u>

This spatial channel of RNN models the spatial dynamics of skeletons. The detailed definition is as follows:

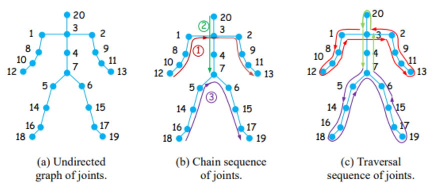

(a) Undirected graph of joints.
(b) Chain sequence of joints.
(c) Traversal sequence of joints.

Figure 4. (a) The physical structure of 20 joints. (b) Convert the joints graph into a sequence. The joints of arms come first, then that of body, finally is that of legs. (c) Use a traversal method to transform the joints graph into a sequence. The order of the sequence is the same as the visiting order of the arrow.

Human body can be considered as an articulated system of rigid segments connected by joints. The physical structure of the 20 joints is represented by an undirected graph in Figure 4(a). Nodes denote the joints and edges denote the physical connections. When an action takes place, this undirected graph displays some varied patterns of spatial structures. For example, clapping is performed with the joints of the two palms striking together, and bending is acted when the joints of the trunk shape into a curve.

To model the spatial dependency of joints, we cast the graph structure into a sequence of joints and exactly develop a relevant RNN architecture. ***The input of the RNN architecture at each step corresponds to the vector of coordinates of a certain joint****. As a joint has only three coordinates, we select a* ***temporal window*** *centered at the time step and concatenate the coordinates inside this window to represent this joint. This RNN architecture models the spatial relationships of joints in a graph structure and is called spatial RNN. (From the paper)*

The important problem here is to convert a skeleton graph to a sequence of joints. And we choose Traversal sequence in our model:

Traversal sequence. As illustrated in Figure 4(c), we first select the central spine joint as the starting point, and visit the joints of the left arm. While reaching an end point, it goes back. Then we visit the right arm, the upper trunk, etc. After visiting all joints, it finally returns to the starting point. We arrange the graph into a sequence of joints according to the visiting order. The traversal sequence guarantees the spatial relationships in a graph by accessing most joints twice in both forward and reverse directions. (From the paper)

According to the experiments done in the paper, when the temporal windows size is one fourth of the total number of frames of one clip, the model performs best. Since in our project, our frame number is 32, the temporal window size we select here is 8.

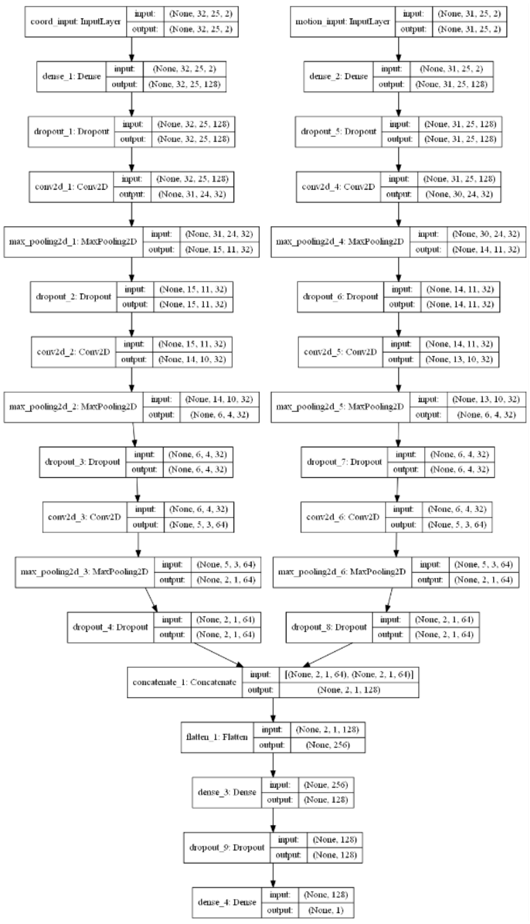
At the end of the spatial RNN, there is also a dense layer with sigmoid activation function.

Weightedsum Layer:

This layer connects the outputs from temporal RNN and spatial RNN, since both RNNs output the possibility of jump, which is between 0 and 1, the Weightedsum layer simply adds them with their weights. According to the paper, the model performs best when we assign weight 0.9 to temporal RNN and 0.1 to spatial RNN.

c) The model from submission3

For submission3, we introduced a model based on CNN. The implementation of this model is based on *Skeleton-based Action Recognition with Convolutional Neural Networks*. It has the following structure:



coord_input: InputLayer | input: (None, 32, 25, 2) | output: (None, 32, 25, 2)
dense_1: Dense | input: (None, 32, 25, 2) | output: (None, 32, 25, 128)
dropout_1: Dropout | input: (None, 32, 25, 128) | output: (None, 32, 25, 128)
conv2d_1: Conv2D | input: (None, 32, 25, 128) | output: (None, 31, 24, 32)
max_pooling2d_1: MaxPooling2D | input: (None, 31, 24, 32) | output: (None, 15, 11, 32)
dropout_2: Dropout | input: (None, 15, 11, 32) | output: (None, 15, 11, 32)
conv2d_2: Conv2D | input: (None, 15, 11, 32) | output: (None, 14, 10, 32)
max_pooling2d_2: MaxPooling2D | input: (None, 14, 10, 32) | output: (None, 6, 4, 32)
dropout_3: Dropout | input: (None, 6, 4, 32) | output: (None, 6, 4, 32)
conv2d_3: Conv2D | input: (None, 6, 4, 32) | output: (None, 5, 3, 64)
max_pooling2d_3: MaxPooling2D | input: (None, 5, 3, 64) | output: (None, 2, 1, 64)
dropout_4: Dropout | input: (None, 2, 1, 64) | output: (None, 2, 1, 64)
motion_input: InputLayer | input: (None, 31, 25, 2) | output: (None, 31, 25, 2)
dense_2: Dense | input: (None, 31, 25, 2) | output: (None, 31, 25, 128)
dropout_5: Dropout | input: (None, 31, 25, 128) | output: (None, 31, 25, 128)
conv2d_4: Conv2D | input: (None, 31, 25, 128) | output: (None, 30, 24, 32)
max_pooling2d_4: MaxPooling2D | input: (None, 30, 24, 32) | output: (None, 14, 11, 32)
dropout_6: Dropout | input: (None, 14, 11, 32) | output: (None, 14, 11, 32)
conv2d_5: Conv2D | input: (None, 14, 11, 32) | output: (None, 13, 10, 32)
max_pooling2d_5: MaxPooling2D | input: (None, 13, 10, 32) | output: (None, 6, 4, 32)
dropout_7: Dropout | input: (None, 6, 4, 32) | output: (None, 6, 4, 32)
conv2d_6: Conv2D | input: (None, 6, 4, 32) | output: (None, 5, 3, 64)
max_pooling2d_6: MaxPooling2D | input: (None, 5, 3, 64) | output: (None, 2, 1, 64)
dropout_8: Dropout | input: (None, 2, 1, 64) | output: (None, 2, 1, 64)
concatenate_1: Concatenate | input: [(None, 2, 1, 64), (None, 2, 1, 64)] | output: (None, 2, 1, 128)
flatten_1: Flatten | input: (None, 2, 1, 128) | output: (None, 256)
dense_3: Dense | input: (None, 256) | output: (None, 128)
dropout_9: Dropout | input: (None, 128) | output: (None, 128)
dense_4: Dense | input: (None, 128) | output: (None, 1)

(click and see detail in browser)

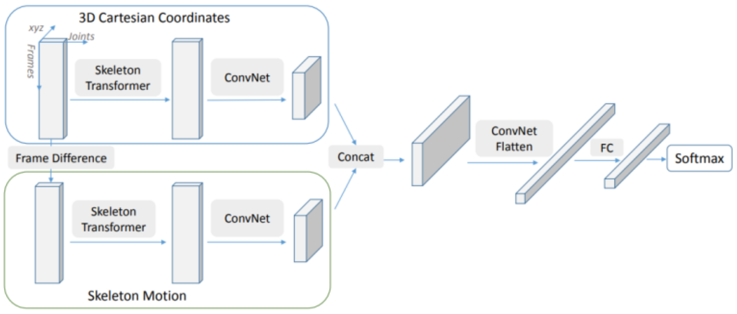


Fig. 1. CNN representation of skeleton sequences for action classification.

<u>3D Cartesian Coordinates stream:</u>

For each video, it has T frames, each frame will generate N joints of skeleton data. For each joint, it has 3 coordinates (X, Y, Z). The way of encoding the skeleton data to an image is:

A skeleton sequence of T frames can be represented as a T ×N ×3 array, which is treated as a T ×N sized 3-channel image.(paper)

In my model, because I used OpenPose, N equals to 25. Besides, for each video, I sample 32 frames from it. Therefore, T equals to 32. Since OpenPose requires two or more camera to generate 3D coordinates, we only have 2D coordinates. Therefore, the input of the Cartesian Coordinates stream in our model is in the shape of (32, 25, 2).

Skeleton Motion stream:

Skeleton Motion stream also takes as input the difference between two consecutive frames, which is very similar to optical flow we have mentioned before. The only difference is that the skeleton motion is computed easily by subtracting the coordination of previous frame from the current frame. Since we sample 32 frames in each video, we will get 31 skeleton motion frames in each video. Therefore, the input of the Skeleton Motion stream is (31, 25, 2).

Skeleton transformer:

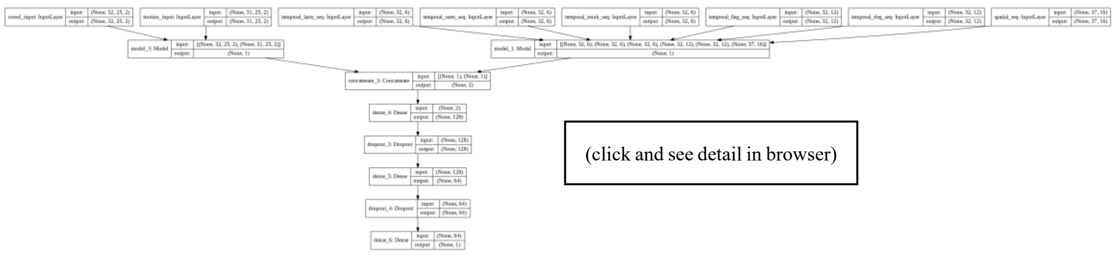
At both streams, after we encode the original skeleton data into images, the first layer is a dense layer. The first dense layer is for skeleton transformer. Its definition is:

For the T × N × 3 skeleton image data, ordering of N joints are arbitrarily chosen (e.g. left eye, right eye, nose, . . .), which may not be optimal. To address this issue, we propose a skeleton transformer module. Given an N × 3 skeleton S, we perform a linear transformation S' = $(S^T \cdot W)^T$, where W is an N ×M weight matrix. S' is a list of M new interpolated joints. Note that both ordering and location of the joints are rearranged. The network selects important body joints automatically, which can be interpreted as a simple variant of attention mechanism.

Skeleton transformer can be implemented simply with a fully connected layer (without bias). We place this module at the very beginning of the network before convolution layers such that it is trained end to end. (paper)

d) Final model: the ensemble of the two models

Then it is time to ensemble the two model, the ensemble method we used is stacking. We use one layer of Concatenate and three layers of Fully-Connected network, to stack the two sub-models. The new model's structure is:


coord_input: InputLayer | input: (None, 32, 25, 2) | output: (None, 32, 25, 2)
motion_input: InputLayer | input: (None, 31, 25, 2) | output: (None, 31, 25, 2)
temporal_larm_seq: InputLayer | input: (None, 32, 6) | output: (None, 32, 6)
temporal_rarm_seq: InputLayer | input: (None, 32, 6) | output: (None, 32, 6)
temporal_trunk_seq: InputLayer | input: (None, 32, 6) | output: (None, 32, 6)
temporal_lleg_seq: InputLayer | input: (None, 32, 12) | output: (None, 32, 12)
temporal_rleg_seq: InputLayer | input: (None, 32, 12) | output: (None, 32, 12)
spatial_seq: InputLayer | input: (None, 37, 16) | output: (None, 37, 16)
model_3: Model | input: [(None, 32, 25, 2), (None, 31, 25, 2)] | output: (None, 1)
model_1: Model | input: [(None, 32, 6), (None, 32, 6), (None, 32, 6), (None, 32, 12), (None, 32, 12), (None, 37, 16)] | output: (None, 1)
concatenate_3: Concatenate | input: [(None, 1), (None, 1)] | output: (None, 2)
dense_4: Dense | input: (None, 2) | output: (None, 128)
dropout_3: Dropout | input: (None, 128) | output: (None, 128)
dense_5: Dense | input: (None, 128) | output: (None, 64)
dropout_4: Dropout | input: (None, 64) | output: (None, 64)
dense_6: Dense | input: (None, 64) | output: (None, 1)

(click and see detail in browser)

4. Hyperparameters:

In this submission, I choose batch size as 256, dropout as 0.5, epochs as 100. For 100 epochs of training, I will always save the best model according to val_loss, as well as the final model after 100 epochs of training.

5. Annotated Code:

Please find my code at my Github(private)
(https://github.com/shyuan7-software/jump_detection_openpose)

6. Training and Testing Performance

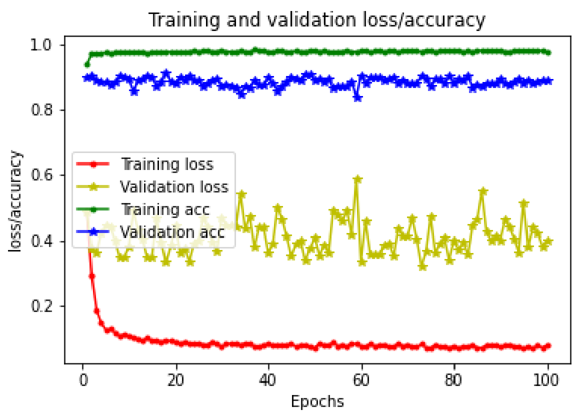
In this section, we only show the performance of our final model: the ensemble model. If you wish to see the performance of the two sub-models, please click: RNN, CNN

a) Testing:

```
Final test acc: 0.9612069129943848

Final test loss: 0.1156773335974792

Best test acc: 0.96875

Best test loss: 0.1035012679881063
```

In first submission, the test acc is 77%, and the second model can have 93% accuracy, and the third model achieves the accuracy of 95%. Now, the latest model has accuracy of 97%, which is the highest accuracy we have achieved so far.

b) Training validation



The pic shows that the model works very well on training dataset, and its performance on validation dataset is also great and stable. The model keeps its accuracy on the validation dataset around 90%. Although the loss is a little unstable, we can only take the model with the lowest validation loss as the best model.

7. Performance in real life:

I made some videos by shooting myself jumping, and use the model to detect jump in the self-made videos. During testing the model, I found both advantages and limitations of my model:

a) The advantages over previous model:

In comparison with the RNN based model in submission2, the ensemble model is smoother. That is, when the person acts like jumping, but does not really jump up (which usually happens when the person tries to jump at the end of the video), the model in submission2 will output 0% or 99%, while the ensemble model will output 20% or 80%, which is smarter.

In comparison with the CNN based model in submission3, the ensemble model is more confident about itself. That is, the CNN based model in submission2 will output many possibilities that are close to 20% or 80%, even there is an obvious jump, while the ensemble model will output more 0% or 99%, which is more confident.

In conclusion, the new ensemble model does not only have higher accuracy and lower loss, in real life application, it also has a better, smarter, and more confident performance, than the previous models.

b) The limitations from OpenPose:

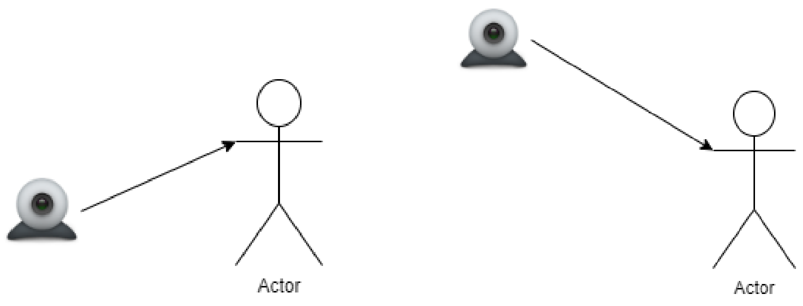
Since the model totally relies on the body landmarks generated by OpenPose as its input, our model will inherit the limitation of OpenPose:

- Non-complete video: if the illumination conditions are not good or not all of the person's body is shown in the video, the model cannot work well, because OpenPose cannot generate precise body-landmarks on the non-complete video.
- Multiple persons: if there are multiple persons showing in the video, the model cannot work well, because when there are multiple persons in the video, the body landmarks generated by OpenPose will become not precise.

c) The limitations from the model itself

Besides the limitation from OpenPose, the model itself also has some limitations:

- The camera needs to be fixed. Suppose we have a person standing still, and the camera itself jumps up and down. Finally, the body landmarks will show that the person is jumping, therefore, the model fails to detect jumping. However, in smart-home application, this is not a limitation, because usually most cameras installed at home are fixed and cannot move.
- The degree of camera will also influence the performance. We have the following images to show the influence:



The left image shows that the camera is in a lower place than the right image. A lower place of camera will bring a better performance, because the change to the y-coordinate is larger when jump happens if camera is placed lower, for the same reason that camera at lower place will make people's legs look longer. Therefore, to get a better performance, it is encouraged to put the camera at a lower place at home.

8. Instruction on how to run your code for training and testing your neural network

a) Install Dependencies

- Python3
- Keras
- TensorFlow
- Numpy
- Matplotlib
- OpenCV
- Googledrivedownloader

b) How to run my code for training

For training a network using my code, just use the following command:

```
python ensemble.py
```

Note that the code on GitHub does not include dataset (because it is too large to upload), so if you clone from my GitHub and run this command, it would fail. If you want to get the dataset, please find it on the appendix of this report.

c) How to run my code for test

- python generate_figure.py [video_path] [body-landmark_directory]

  e.g. python generate_figure.py test_video1.mp4 test_video1

  test_video1.mp4 is a MP4 video file

  test_video1 is a directory containing all body-landmark json-files of test_video1.mp4, generated by OpenPose

- Check results:

  After running generate_figure.py, check your results at the 'OUTPUT' directory

- More details:

  Please find this video (private) (https://youtu.be/RItYYhAImLs)

  If you want to get the sample videos and their landmark files, please find them on the appendix of this report.

Appendix

For your test convenience, I provide some sample videos and their landmark files as following:

sofa.mp4	https://youtu.be/Il0c4LaDsQI
raw_video.mp4	https://youtu.be/sOZx88CmMOE
no_jump.mp4	https://youtu.be/wdpJLK31rIY
keep_jump.mp4	https://youtu.be/xO2b17cI9Wo
intermedia.mp4	https://youtu.be/1KJj8GDu0d4
bed.mp4	https://youtu.be/JiuroZvrt8M
side_small_power.mp4	https://youtu.be/Xkee64oEeBM
side_full_power.mp4	https://youtu.be/SoTE9ZN9QKI
front_small_power.mp4	https://youtu.be/j1_AP10X798
front_full_power.mp4	https://youtu.be/hh1LJGicr8U
back_small_power.mp4	https://youtu.be/2XeIm2Hywd4
back_full_power.mp4	https://youtu.be/QZUxwz2MYOY
random.mp4	https://youtu.be/J6nTsPEfElQ
landmark files	https://drive.google.com/file/d/1TO9qZnFNA0U0Kj7CNqGJTSWKzFSaqOC5/view?usp=sharing

If you want to access my dataset (train, valid, and test), please access it through google drive:

Clips and Landmark files	https://drive.google.com/open?id=1zJWq8R9FKW6VNq2yfp4GrVmTuFabQU32
.NPY files (for fast loading dataset)	https://drive.google.com/open?id=1x4Lvsm_ElDLu17BdMXlPEKStYQ13FnOq