# Jump Detection Project Report - Third Submission

# April 2020

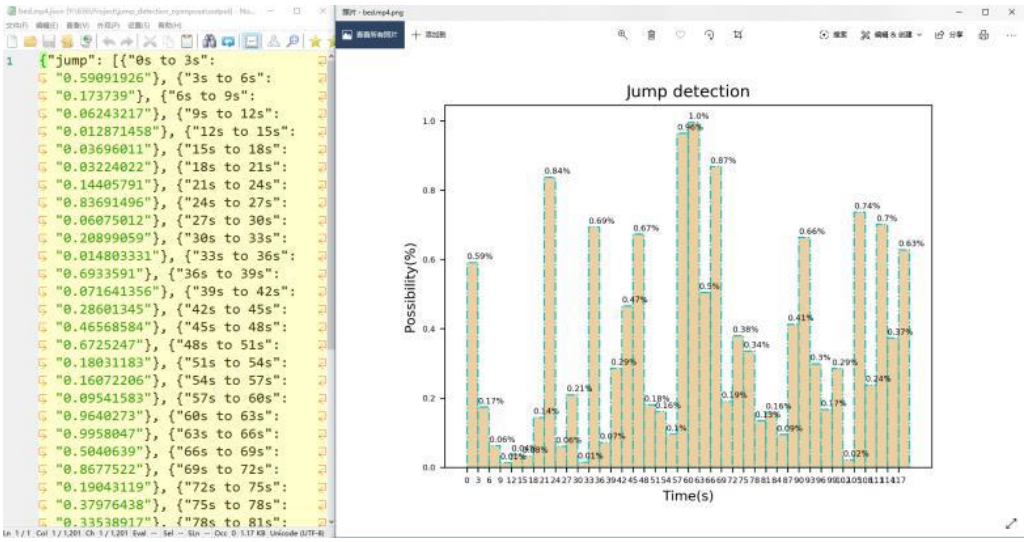
## 1. Topic

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:




## 2. Work and Improvement since second submission

a) Work and experiments:

In our second submission, we chose to disregard the information from RGB frames, and only consider the information from skeleton body-landmarks. And we found several problems of totally relying on the skeleton body-landmarks as input:

Angle of camera: The change of camera angle makes it difficult for OpenPose to extract human body-landmark skeleton data very precisely.

Different kinds of jump: When a person jumps only using his tiptoes, instead of using his legs' strength, the height of jumping is low, and the change to the skeleton data is also small.

For the above two reasons, the model which totally relies on skeleton data input is hard to detect jump when the angle of camera changes while jump, or when people jump without

using their legs' strength.

In order to solve the problems, I decided to involve a CNN based network in our model, which takes original RGB frames as input. Actually, we have done lots of experiments about CNN based model in our second submission: Conv2D & RNN model, pre-trained CNN & RNN model, and Conv3D model, but none of them work well. After doing some research, I found that that it is difficult to train CNN and RNN at the same time. In order to train CNN and RNN well, the CNN needs to be firstly trained so that it can distinguish human and background in images, and then RNN needs to learn the meaning of CNN output, both of which are tough tasks. Therefore, the combination of CNN and RNN is not popular in the human action recognition field. Instead, recent years people tried to only use CNN to classify human action. In the paper of *Two-Stream Convolutional Networks for Action Recognition in Videos*, they choose to only use Conv2D layer to analyze the spatial information and temporal information of a video. The structure is as follows:

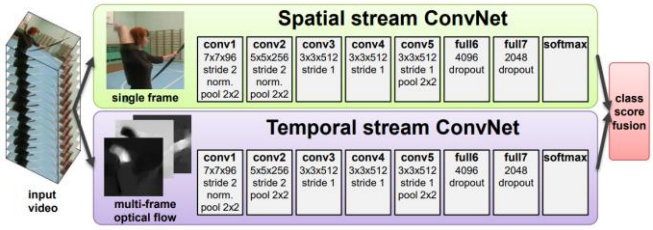



Figure 1: **Two-stream architecture for video classification.**

The model does a pre-processing for each video, and transfer a video to a single frame of RGB image and multi-frames of optical flow. The RGB image is used to analyze the spatial feature of the video, and the multi-frames of optical flow is used to analyze the temporal feature.

The input of spatial stream is simple, it takes one frame from one video as its input.

The input of temporal stream is complicated, it takes the stack of multiple frames of optical flow from each video as its input.

The optical flow is as following:

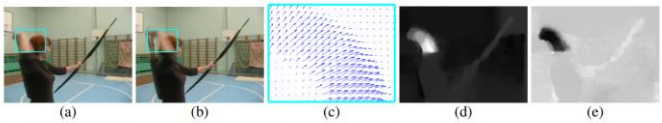



Figure 2: **Optical flow.** (a),(b): a pair of consecutive video frames with the area around a moving hand outlined with a cyan rectangle. (c): a close-up of dense optical flow in the outlined area; (d): horizontal component $d^x$ of the displacement vector field (higher intensity corresponds to positive values, lower intensity to negative values). (e): vertical component $d^y$. Note how (d) and (e) highlight the moving hand and bow. The input to a ConvNet contains multiple flows (Sect. 3.1).

One frame of optical flow is generated by two consecutive frames in a video. It will indicate the distance and direction of movement of each point from previous frame to the next frame.

One frame of optical flow can then be divided to two frames of optical flow. One indicates the movement in direction X, another one indicates the movement in direction Y.

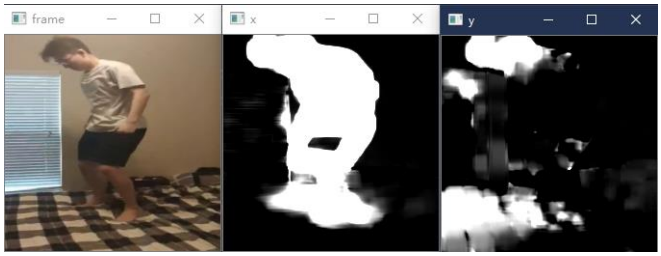
Then, if a video has L frames, it can generate 2(L-1) frames of optical flow. According to the paper, in order to input the 2(L-1) frames into CNN, they simply stack them. For example: assume we have a video having 31 frames, then the shape of the frames of optical flow will be (60, 224, 224, 1). Among the shape, 60 is 2(L-1), the number of frames; 224 is the height and the width of each frame, and 1 means one pixel in each frame only has one value, because as we can see from Figure 2, the optical flow is black-white image. After stacking, the shape will

become (224, 224, 60). The details of stacking are as follows:

> **Optical flow stacking.** A dense optical flow can be seen as a set of displacement vector fields $\mathbf{d}_t$ between the pairs of consecutive frames $t$ and $t+1$. By $\mathbf{d}_t(u, v)$ we denote the displacement vector at the point $(u, v)$ in frame $t$, which moves the point to the corresponding point in the following frame $t+1$. The horizontal and vertical components of the vector field, $d_t^x$ and $d_t^y$, can be seen as image channels (shown in Fig. 2), well suited to recognition using a convolutional network. To represent the motion across a sequence of frames, we stack the flow channels $d_t^{x,y}$ of $L$ consecutive frames to form a total of $2L$ input channels. More formally, let $w$ and $h$ be the width and height of a video; a ConvNet input volume $I_\tau \in \mathbb{R}^{w \times h \times 2L}$ for an arbitrary frame $\tau$ is then constructed as follows:
>
> $$I_\tau(u, v, 2k-1) = d^x_{\tau+k-1}(u, v), \quad (1)$$
> $$I_\tau(u, v, 2k) = d^y_{\tau+k-1}(u, v), \quad u = [1; w], v = [1; h], k = [1; L].$$
>
> For an arbitrary point $(u, v)$, the channels $I_\tau(u, v, c), c = [1; 2L]$ encode the motion at that point over a sequence of $L$ frames (as illustrated in Fig. 3-left).

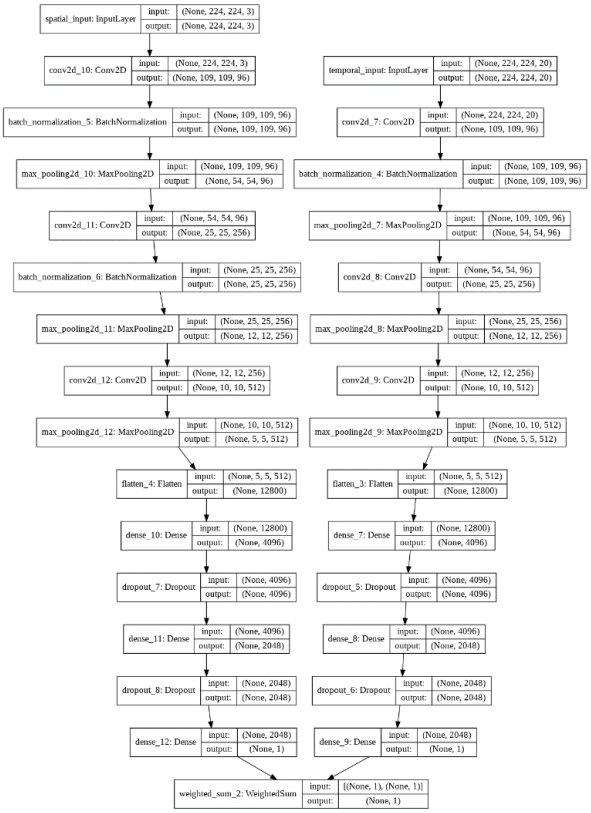
From the paper

Then I started working on the new CNN model based on the paper. First, I used OpenCV to extract optical flows of my dataset. The output optical flows are:

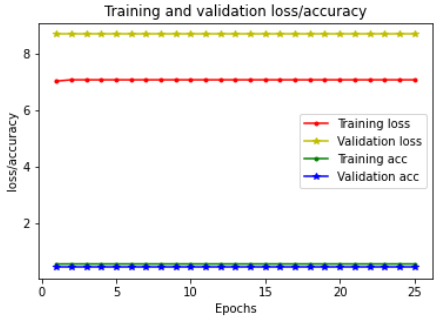



For each video, I sample 10 frames of optical flow from it, as the temporal input of the CNN based model, and resize them to (224, 224). Therefore, the temporal input in my model is (batch size, 224, 224, 20). And the spatial input in my mode is (batch size, 224, 224, 3).

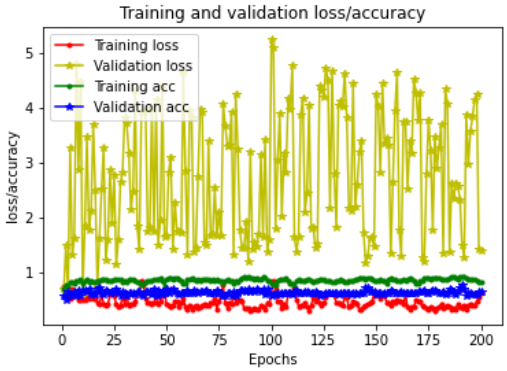
As described in the paper, I construct my model as follows:

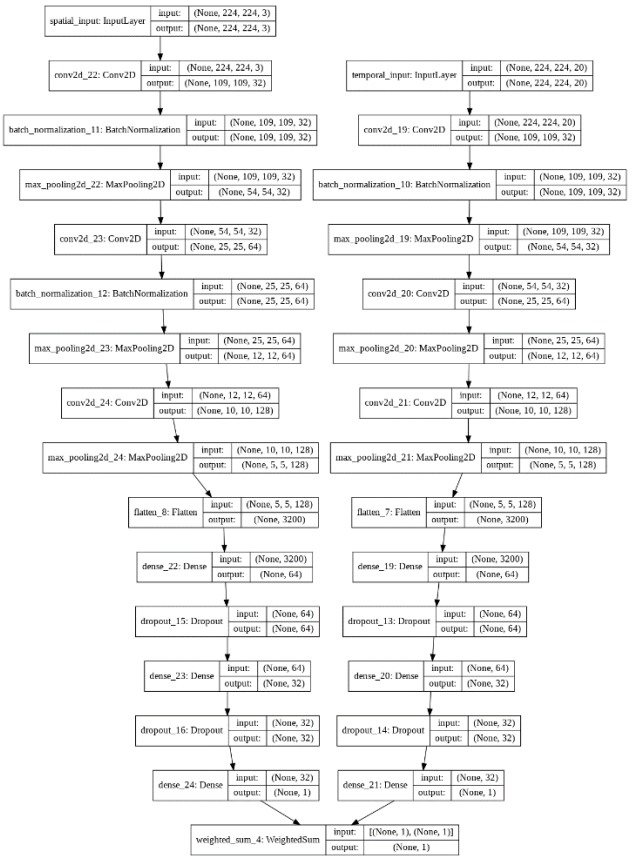


However, the result is not good:




Then I try to modify the hyper-parameter, number of neurons, the best model and result I got is as follows, which is not good.



Obviously, the model described by the paper does not work well in detecting jump. First, the spatial stream actually does not work for detecting jump. In the paper, the dataset it uses to classify human actions include football and basketball, which are actions that has very unique feature, for example lawn is a feature of football, wooden ground floor is a feature of basketball. However, jump action is much more general than those actions, which can happen anywhere. And usually when the dataset is small, CNN will also learn the background of each frame, make it hard for the model to learn the feature of the distance between human and ground. Second, the noise of optical flow is too large to ignore. When camera moves, the background will also generate optical flow, which is a noise for model. Therefore, the temporal stream also does not work well.

b) Improvement and contribution:

Then I found another CNN based model, which takes skeleton data as input, the model is based on *Skeleton-based Action Recognition with Convolutional Neural Networks*. The result of this model is much better than the model I mentioned above, and even better than the two-stream RNN model in second submission.

This CNN based model is the main contribution and improvement of this submission. It has the highest accuracy among all models we get. In the meantime, it is very easy to train: I can train the model even with NVIDIA GeForce MX250 Graphics Card. I will give detailed introduction to the model in the Network Model section.

# 3. Dataset

a) Raw dataset: video

I use part videos from Weizmann, UCF-101, STAIR-actions. These are all datasets including different human actions (jump, run, walk, sit down, exercise, etc.). I collect around 900 videos from these datasets. Half of their topics are jump, another half videos' topics are others (run, walk, sit down, exercise, stand up).

Besides, I also shoot myself jumping or not to 6 videos, each of them has 2 mins time-length.

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:

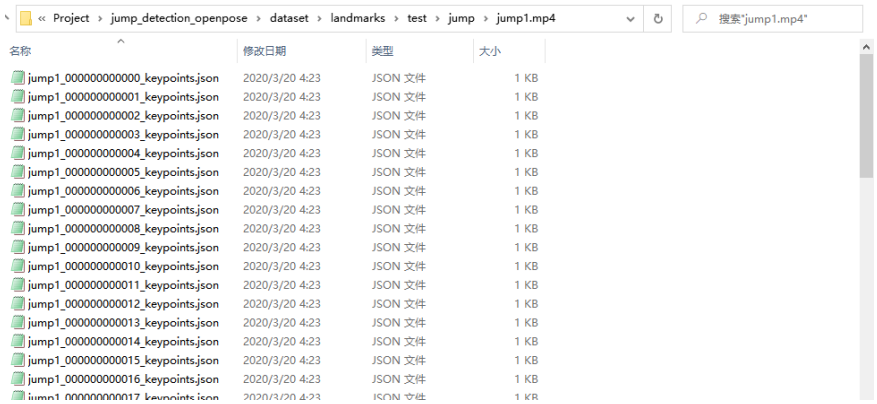
```
Y:\636\Project\jump_detection_openpose\dataset>tree
卷 Work 的文件夹 PATH 列表
卷序列号为 E856-3BE6
Y:.
├─clips
│  ├─test
│  │  ├─jump
│  │  └─others
│  ├─train
│  │  ├─jump
│  │  └─others
│  └─valid
│      ├─jump
│      └─others
└─landmarks
    ├─test
    │  └─jump
    │      ├─jump1.mp4
    │      ├─jump10.mp4
    │      ├─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/”. Clips are stored in “dataset/clips/”, while landmark files are stored in “dataset/landmarks/”.

We get 900 videos from Weizmann, UCF-101, STAIR-actions, these videos are split into around 1800 clips. We choose 1600 of them as train clips, 200 of them as valid clips. I also get 6 videos by shooting myself using smartphone. These 6 videos are split to 200 clips, and we use the 200 clips as test clips.

For each video, there exist a directory under “dataset/landmarks/” that has the same name as the video 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:




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

## d) Load dataset:

There are two ways of loading dataset:

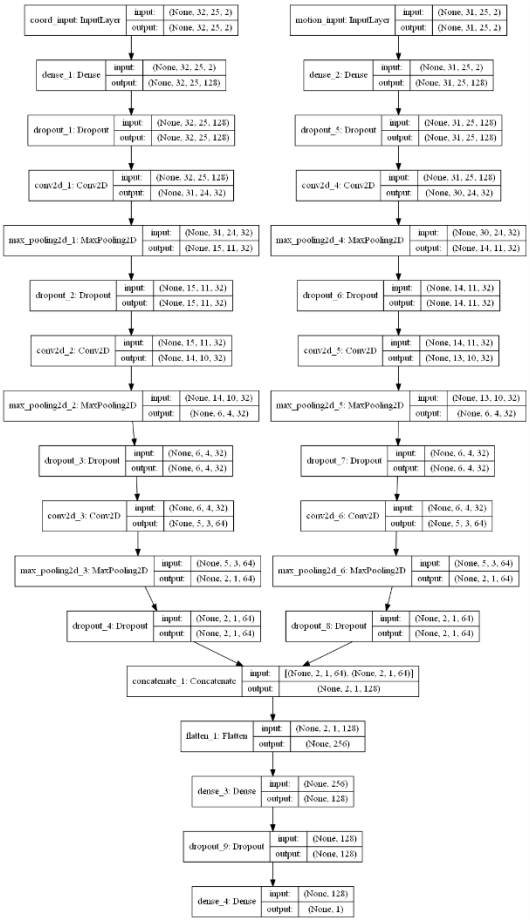
The first way is to use opencv to load clips, and json python library to load landmark files. However, this is very slow, because we have so many files to read and I/O operation is much slower than CPU, so we have the second way:

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:


```
import numpy as np
train_videos = np.load("10image/train_videos.npy")
train_tracks = np.load("10image/train_tracks.npy")
train_lables = np.load("10image/train_lables.npy")
valid_videos = np.load("10image/valid_videos.npy")
valid_tracks = np.load("10image/valid_tracks.npy")
valid_lables = np.load("10image/valid_lables.npy")
test_videos = np.load("10image/test_videos.npy")
test_tracks = np.load("10image/test_tracks.npy")
test_lables = np.load("10image/test_lables.npy")
```

# 4. Network Model

a) Overview:



This model is designed based on *Skeleton-based Action Recognition with Convolutional Neural Networks*. It encoded skeleton sequence data to image as input. A more straightforward description of this model is as follows:

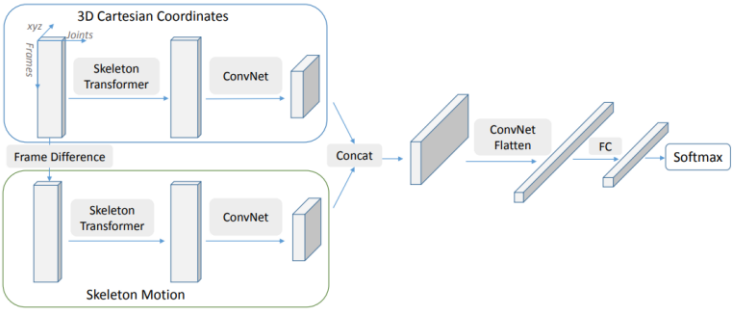



**Fig. 1**. CNN representation of skeleton sequences for action classification.

b) 3D Cartesian Coordinates stream:

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).

c) The 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).

d) 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)*

## 5. Hyperparameters:

In this submission, I choose batch size as 64, dropout as 0.5, epochs as 200. For 200 epochs of training, I will always save the best model according to val_loss, as well as the final model after 200 epochs of training.

## 6. Annotated Code:

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

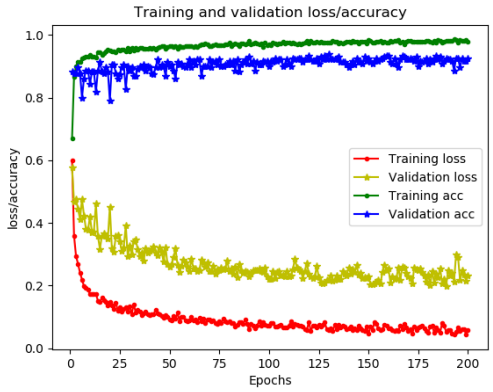
## 7. Training and Testing Performance

a) Testing:

```
Final test_acc:0.9482758641242981
Final test_loss:0.257728643458465
Best test_acc:0.9525862336158752
Best test_loss:0.2370875287672569
```

In first submission, the test acc is 77%, and the second model can have 93% accuracy, and the latest model achieves the accuracy of 95%, 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 stable and gets better and better. The model works best (lowest val_loss) around 180 epochs. Besides, the model has fewer than 100, 000 parameters to train, making the training very fast and easy.

## 8. Performance in real life:

I made some videos by shooting myself jumping, and use the model to detect jump in my video. The follows are some interesting discoveries:

In comparison with the two-stream RNN model in second submission, the two-stream CNN model is less confident, meaning that it will not output too many 0% or 100%, instead, it will output more possibilities that are close to 20% and 80%. Besides, it can detect some jumps that RNN model fails to detect, but it also fails on detecting some jumps that RNN models succeed to detect. Therefore, we can see that the CNN model as a supplementary of RNN and vice versa.

The problems of angle of camera and different kinds of jump remain unsolved, because the source cause of the problems is the limitation of OpenPose: it cannot generate precise data when the

camera angle changes in video; When people jump without using the strength of their legs obviously, the change to the skeleton data is small.

For non-complete video, which means that 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 work well on the non-complete video. And since our model is built totally based on OpenPose, non-complete video will become a bottleneck of our model.

# 9. Future improvement and challenge:

From the section 8, we made a conclusion that CNN model is a supplementary of RNN model, so if we can combine them, we can get a better model that will detect jump more precisely. Therefore, one direction is to find an optimal way of fuse the outputs from two model together, and transform the two outputs to the final possibility of detecting jump.

After we fuse the two model, we will have a relatively good skeleton-based network architecture. However, the problems of skeleton based model, caused by the limitation of OpenPose, still remain unsolved. To overcome the problems, We need another sub-network which does not rely on OpenPose. I think the simplest model is the best model. I will train a CNN model and let it detect whether human's feet are touching the ground. If it works, I will include it as a sub-stream in my model, if not, we can fairly say that it is very difficult to extract the feature of the distance between feet and ground based on original RGB video input, making it hard for the model to detect jump and solve the problems introduced by OpenPose.

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

a) Install Dependencies
   - Python3
   - Keras
   - TensorFlow
   - Numpy
   - Matplotlib
   - OpenCV

b) How to run my code for training

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

```
python cnn_model.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

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/file/d/1shPnXQeDR2yWOFankFqCrR7JyakFenSl/view?usp=sharing |
| .NPY files (for fast loading dataset) | https://drive.google.com/drive/folders/1b11D5WAf7ELt4FV2HNGCvS3ZCKkNhYJS?usp=sharing |