3D Body Tracking using Deep Learning

THESIS

A thesis submitted in partial
fulfillment of the requirements for
the degree of Master of Science in
the College of Engineering at the
University of Kentucky

By
Qingguo Xu
Lexington, Kentucky

Director: Dr. Ruigang Yang, Professor of Computer Sciences
Lexington, Kentucky 2017

ABSTRACT OF THESIS

3D Body Tracking using Deep Learning

This thesis introduces a 3D body tracking system based on neutral networks and 3D geometry, which can robustly estimate body poses and accurate body joints. This system takes RGB-D data as input. Body poses and joints are firstly extracted from color image using deep learning approach. The estimated joints and skeletons are further translated to 3D space by using camera calibration information. This system is running at the rate of 3 4 frames per second. It can be used to any RGB-D sensors, such as Kinect, Intel RealSense [14] or any customized system with color depth calibrated. Comparing to the sate-of-art 3D body tracking system, this system is more robust, and can get much more accurate joints locations, which will benefits projects require precise joints, such as virtual try-on, body measure, real-time avatar driven.

KEYWORDS: 3D body tracking, pose estimation, joints detection, deep learning, Caffe, multiple thread programming

Author's signature: Qingguo Xu

Date: August 2, 2017

3D Body Tracking using Deep Learning

By
Qingguo Xu

Director of Dissertation: Dr. Ruigang Yang

Director of Graduate Studies: Dr. Miroslaw Truszczynski

Date: August 2, 2017

ACKNOWLEDGMENTS

To my life-coach, my adviser, Dr. Ruigang Yang, a man who helps me a lot during my master program. He is always very nice and patient to me, especially when I don't know what to do with the project. He came up a few projects for me to choose. I really appreciate his kindness and consideration for me. Besides the study life, Dr. Yang also helps me find several on-campus jobs, which is so good for a student to pay bills. In my mind, Dr. Yang is the example of a good adviser and a professor.

To my committee members, they all point out some problems in my project, and give me some advice, like I should skip some frames if I want to do the live demo. Those suggestions can also benefit me in the future.

To the DGS of our department, Dr. Miroslaw Truszczynski, a good man with a lot of patience. As an international student, I met some status issue, and Dr. Truszczynski always help me out of it. He is a very helpful DGS for all the students in our department, especially for the international students.

To the most important person in my life, Yajie Zhao, she is so supportive and always believe in me. She helps me so much that I cannot list all of them. She is the most helpful person in my life.

To all other workmates in our group, I want to thank all of you for your nice and kindness. I will miss the time when we group together, talk and discuss.

Thanks for all your encouragement!

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1 Introduction

Human body tracking is an important research area in computer vision, which has many applications, like human-computer interaction, gaming, making movie, security, telepresence, view synthesis. Virtual reality (VR) and augment reality (AR) are developing very fast in recent few years, which also need body tracking. At the beginning, human body was treated as a whole component, which can only track where the person goes. Now, we also want to know what the person is doing. So it is needed to recognize the person's poses and actions.

Nowadays, there are many ways to track a human body. Kinect sensor can track up to 6 people at a time and can track 2 users in details like skeletal joints and orientations, and hand states. A RGB color camera and 3D depth sensor (including infrared camera and infrared emitters) are built in the Kinect. Kinect first compute depth map from the infrared image, and then detect body joints by using a randomized decision forest, learned from over 1 million training examples [29].

Although Kinect can track human body in real time, it cannot tell whether a person in the view is facing or showing back to the camera. So in practice, this shortcoming will restrict the popularity in applications that user may turn around. Besides that, Kinect cannot detect the joints very accurately. In figure 1.1, the left image shows Kinect will fail when user turn around, and the right image shows the two knees are not detected very accurate.

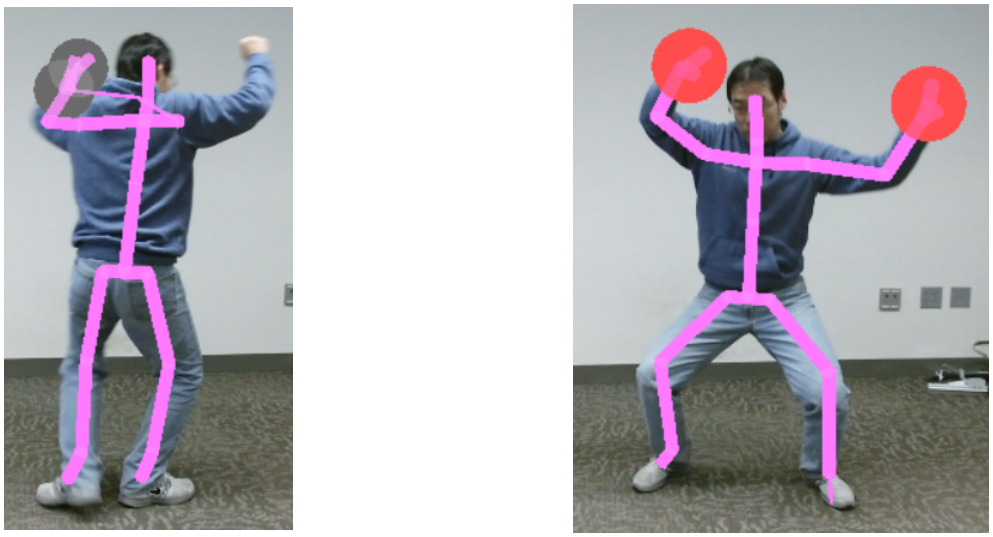

Figure 1.1: Joints detection example by Kinect.

Cao *et al.* [9] present a near real time pose estimation using deep learning. It can produce much more accurate joints detection results, but it's 2D based body tracking. Obviously, 3D body tracking is more useful than 2D tracking. Many applications, like view synthesis and motion sensing games, can benefit from accurate 3D body

tracking. Therefore, I want to build up a 3D body tracking system which can detect human body joint more accurate than Kinect.

There are many existing body tracking techniques [35, 26, 24, 10, 7, 4]. . I choose to use the deep learning method presented in [9], because it can produce stae-of-art joints detection results and it is faster than others.

Inspired by [9], I want to detect joints in 2D space and then convert the detected results into 3D. I choose Kinect as the input device, because Kinect can provide 2D color images which can be used for joints detection and depth images which can be used for 3D model reconstruction.

The thesis is organized in the following order. Chaper 2 introduces all the background technologies that used in the system. Chapter 3 simply shows the how to design and setup the system. In Chapter 4, a lot of implementation details are discussed. Chapter 5 shows some results of the system and Chapter 6 conclude the thesis.

Chapter 2 Related work

Although body tracking has many applications in computer vision and many existing techniques, it is still not easy to track the body very accurately.

Some motion capture (mocap) and tracking systems can provide accurate tracking result, but will need special markers, like OptiTrack[25]. Those maker-based system do have some strengths, such as accuracy, reliability and speed. But it has more drawbacks. First it will need special makers and suits, if some markers are occluded, the results cannot be guaranteed. The system setup is also very expensive. The system need careful calibration by experts.

Modern markerless motion capture use advanced computer vision technology to identify and track subjects without the need for any markers. There are some top-down methods [31, 12, 27] to employ a person detector first and then perform single-person pose estimation. Since there are existing single-person pose estimation techniques [35, 26, 24, 10, 7, 4], those top-down methods can directly use them. Besides the bottleneck of the pose estimation, those top-down methods depend on the accuracy of person detector. If person detector fails, they cannot track the human body successfully.

Body tracking can be divided into 2 categories: 2D-based body tracking and 3D-based body tracking. At first, body tracking was applied on 2D images and videos sequences. There are many techniques that can be used for body joints detection and tracking. [17, 22, 3] take advantage of Kalman filter. [28] can estimate 2D human pose from video using optical flow. While recently, deep learning is also used for pose estimation and body tracking [5, 9]. The advantage of using deep learning is it can produce accurate and near realtime result.

3D-based body tracking has much more applications and more useful. Many previous research are focus on 3D body tracking. Mao *et al.* [36] can estimate 3D pose very accurately just using a single depth image. A set of pre-captured motion exemplars are need to be matched with the input depth image. It will first estimate a pose and then refine it by directly fitting the body configuration with the input depth. Roland *et al.* [21] present a method that use stochastic sampling to track full body from multiple view. A volumetric reconstruction of a person will be extracted from silhouettes in multiple video images. Then an articulated body model will be fitted to the data with stochastic meta descent optimization. Cheung *et al.* [11] present a shape-from-shilhouette method for body tracking. Colored surface points are used to segment the hull into rigidly body parts, based on the results of the previous frames. The constraint of equal motion of parts at their coupling joints are used for estimating joints positions.

Although 3D body tracking is much more useful than 2D, the computing cost is also more expensive than 2D tracking. To reduce computing cost, deep learning based methods are introduced for body tracking and pose estimation. Both [33] and [16] estimate human pose using convolutional networks. Arjun *et al.* from [16] also present that a specific variation of deep learning is able to outperform all existing

traditional architectures on this task. [9] can produce very accurate joints position on 2D images using deep learning and it is close to realtime. All these works are using deep learning on 2D images. Deep learning should also work on 3D body tracking, but the biggest problem is that there is no such 3D pose database that can be used to train deep learning network.

Chapter 3 Background

3.1 Convolutional Neural Network (CNN)

Deep learning is a class of machine learning algorithm that have many layers and getting more and more attention these days. Although it may takes very long time to train a meaningful model, deep learning can provide very impressive results. Convolutional Neural Network (CNN) is the most basic and popular deep learning network. CNN is used in this project, so I will talk about more details about CNN. Before talking about CNN, I need to introduce ANN first. Artificial Neural Network (ANN) is developed to help computer "think" as a human being. An ANN consists of nodes in different layers: input layer, hidden layer and output layer. Every basic unit in the network is called a neuron, or a node. The neuron can receive input from external source (in input layer) or some other neurons (in hidden/output layer) and can compute an output[19]. Figure 3.1 shows a single neuron takes inputs X1 and X2 with weights w1 and w2 respectively. Additionally, there is another input 1 with weights b, which is called bias. The output is computed as $Y = f(w1.X1 + w2.X2 + b)$ and the function f is called activation function and it's non-linear. The purpose of function f is to introduce non-linear into the output, since most real world data is non-linear. There are 3 most common non-linear functions: Sigmoid, tanh, and ReLU (Rectified Linear Unit). Figure 3.2 shows the 3 activation functions. Sigmoid takes a real-valued input and maps it to range between 0 and 1. tanh function squashes the input to the range [-1,1]. ReLU threshold the input value at zero. ReLU is the one I use and its equation is as follow.

$$f(x) = max(0, x)$$

Convolutional Neural Networks are a category of Neural Networks that are proven very effective in image recognition and classification[20]. CNN have been successful in identifying faces, objects and traffic signs. Figure 3.3 shows a simple CNN and classifies an input image into four categories (outputs). There are 4 main operations in Figure 3.3: convolution, ReLU, pooling and classification (Fully connected layer).

In this thesis, I introduce 2D convolution applied on 2D images. Every image can be considered as a 2D matrix of pixel values and the pixel value range from 0 to 255. To apply convolution on image matrix, a small 2D matrix (called kernel or filter) is needed. Every time the filter scan the image matrix, we can get a convolved image matrix, which is called feature map. If we apply different filters on the same image, we can get different feature maps. Therefore, the more filters, the more image features can be extracted and the better performance the CNN can perform. In practice, a Convolutional Neural Network can learn the values of filters on its own during the training process[20]. During the convolution process, the size of feature map is controlled by 3 parameters[34]: depth, stride and zero-padding. Depth is the

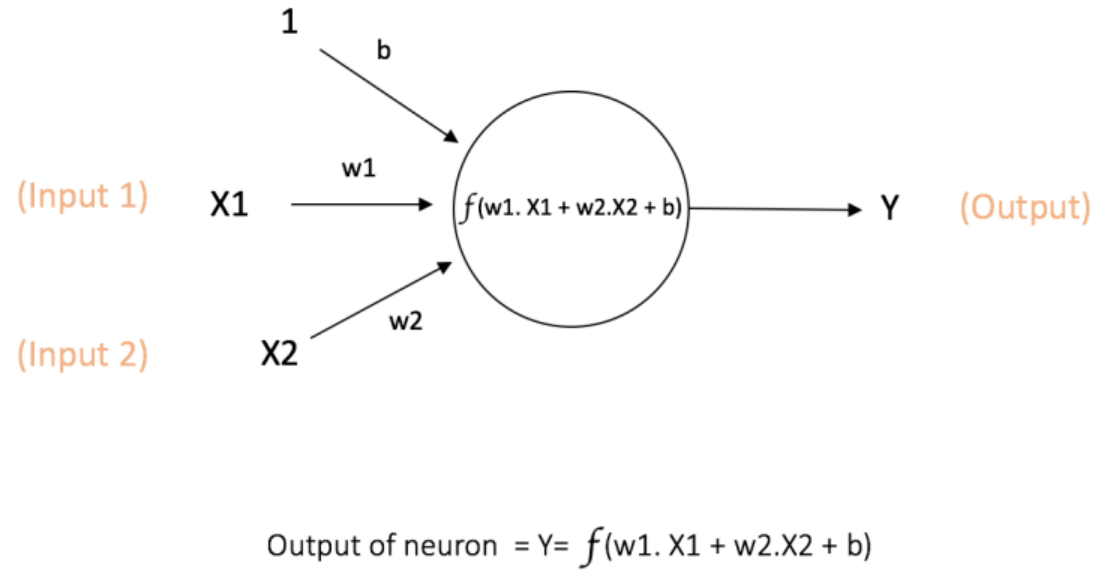

Figure 3.1: A single neuron. Source[19]

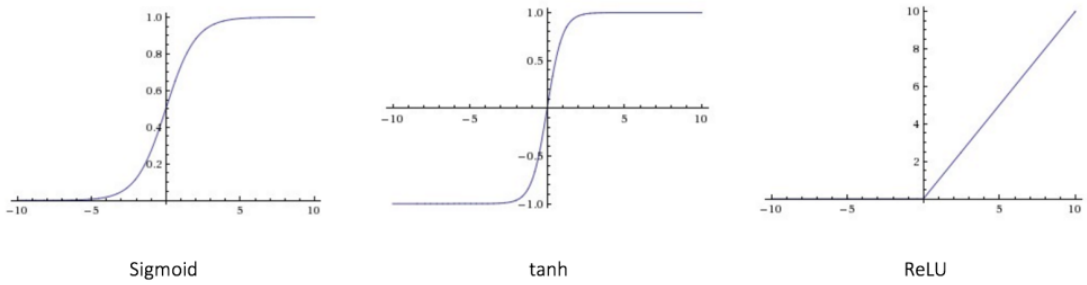

Figure 3.2: Different activation functions. Source[19]

number of filters used for convolution operation. As discussed above, more filters means more feature maps; Stride is the number of pixels by which we slide the filter. Sometime, we need to apply filters to bordering elements of the input images, so it's very convenient to pad zeros around the input images' border.

ReLU stands for Rectified Linear Unit and it's a non-linear operation as shown in Figure 3.2. This is an element wise operation and it will replace all negative values with zero. After the ReLU operation applied on feature map, the output is Rectified feature map and non-linearity is introduced in the CNN.

Pooling is also called subsampling or downsampling. It reduces the dimension of each (rectified) feature map but retains the most important information. There are many different types of pooling, such as Max, Average, Sum etc. In practice, Max pooling has been shown to work better and it's the pooling method I use. Figure 3.4 can show the process of Max pooling very clearly.

At the end of CNN are the fully connected layers. The fully connected layer

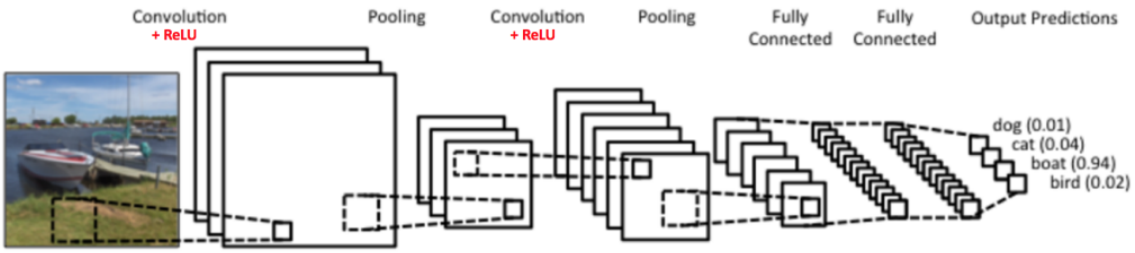

Figure 3.3: A simple CNN. Source[20]

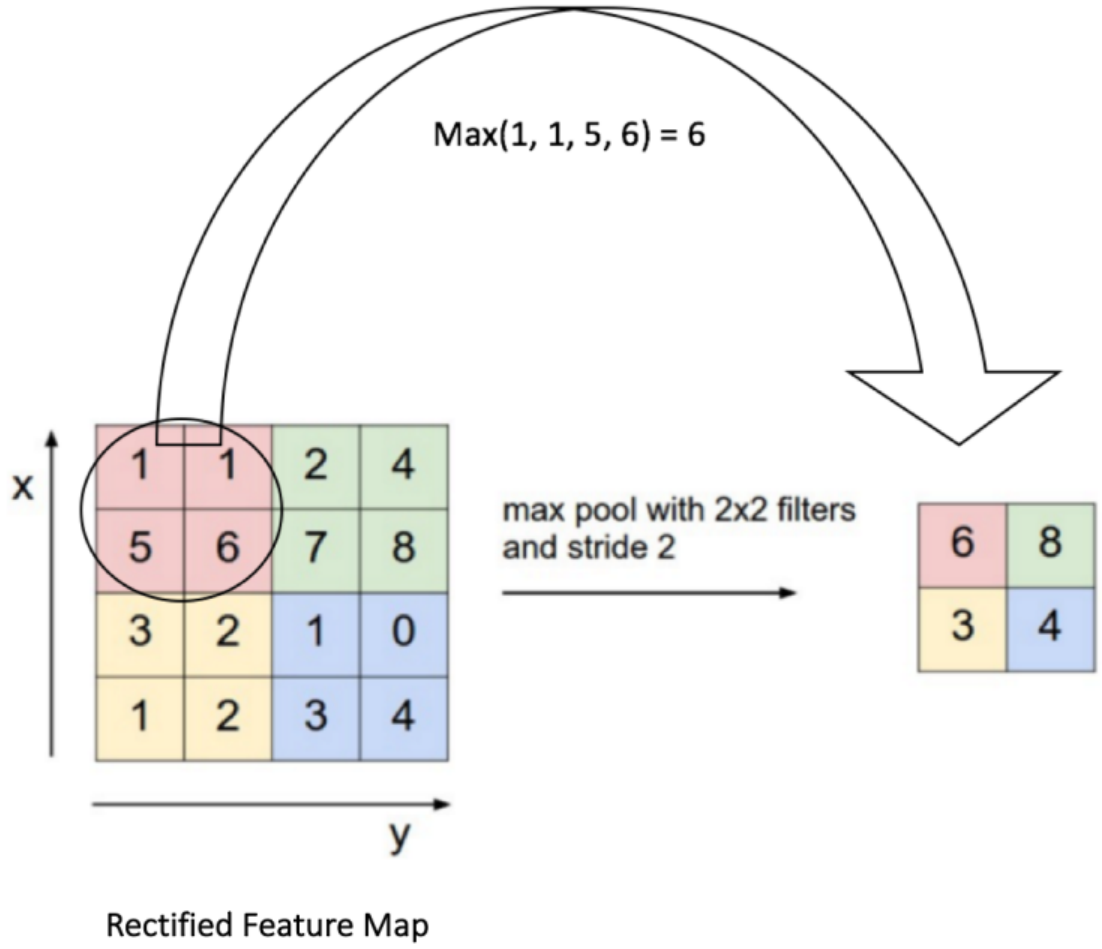

Figure 3.4: Max pooling. Source[34]

is a traditional Multi Layer Perceptron (MLP) which contains one or more hidden layers. For the fully connected layer, every neuron in the previous layer is connected to every neuron on the next layer. The purpose of the fully connected layer is to use the outputs of convolutional and pooling layers for classifying the input image into various classes based on the training dataset. Besides classification, adding a fully connected layer in CNN is also a cheap way to learn non-linear combinations of the

features from convolutional and pooling layers. Combinations of those features might perform even better than use those features directly.

Hours Studied	Mid Term Marks	Final Term Result
35	67	1 (Pass)
12	75	0 (Fail)
16	89	1 (Pass)
45	56	1 (Pass)
10	90	0 (Fail)

Figure 3.5: Student marks (a backpropagation example). Source[19]

3.2 Backpropagation

Backpropagation is used in ANN to calculate the gradient of the loss function with respect to the weights [2]. Backward propagation of errors is one of the several ways in which an artificial neural network (ANN) can be trained. It's a supervised training, which means, it learns from labeled training data. The labeled data works as the groundtruth, so it's like a supervisor guiding ANN to learn from errors. Because for each input in the training dataset, the output is known.

How does the Backpropagation algorithm work? First, all the edge weights are initialized randomly. Then for every entry in the training dataset, the ANN is activated and its output will be compared with the known output. The error will be propagated back to the previous layers to calculate the gradients. Then we can use an optimization method such as Gradient Descent to adjust all the weights in the network aiming to reduce the error at the output layer. This process will be repeated until the output error is small enough or below a predetermined threshold.

There is a good example to help understand backpropagation. There is a student marks datasheet in Figure 3.5. Now I need to predict whether a new coming student with 25 hours study and 70 midterm marks will pass or fail.

We can use follow equation to calculate the output, which f is an activation function.

$$V = f(1 * w1 + 35 * w2 + 67 * w3)$$

Suppose we get the outputs of 2 output layer nodes are 0.4 and 0.6 respectively. Figure 3.6 shows the process. The outputs are far from desired probabilities (1 and 0).

Since there are 5 entries, we will calculate the total error at the output layer. Then we will propagate these errors back through the network using backpropagation to calculate gradients. We can use gradient descent to adjust all the weights (new weights are $w3, w4, w5$). By doing so, the error at the output layer will reduce. Figure 3.7 shows the procedure.

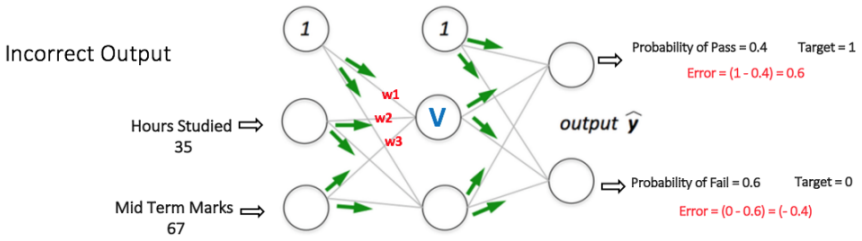

Figure 3.6: Forward propagation step. Source[19]

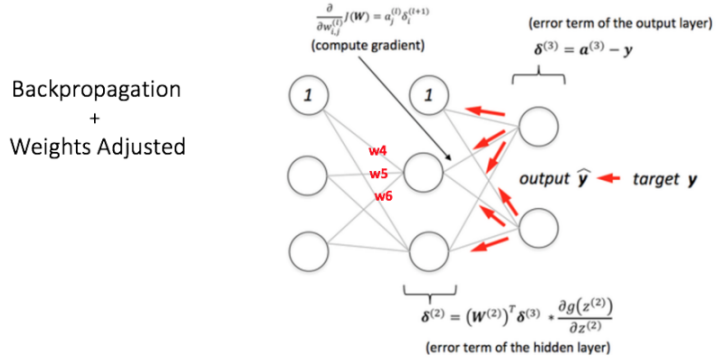

Figure 3.7: Backward propagation step. Source[19]

After backpropagation, we can use equation $V = f(1 * w1 + 35 * w2 + 67 * w3)$ to calculate output and the output error of output layer will be reduced. Like shown in Figure 3.8, the output are closer to the desired output than the first time. If the error are acceptable, then the network is trained and we can use it to predict the new coming student's final grade.

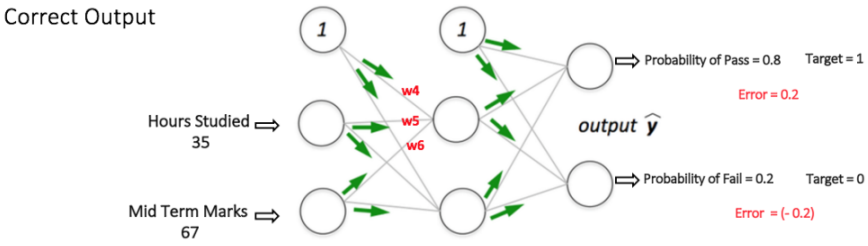

Figure 3.8: After update weights. Source[19]

3.3 Caffe

Caffe[18] is a deep learning framework and it's developed by Berkeley AI Research (BAIR)/The Berkeley Vision and Learning Center (BVLC). Models and optimization are defined by configuration without hard-coding[8]. So only with a few modification in the configuration, a different neural network will be created. There is a single flag

to help switch between CPU and GPU. So it can be trained on a GPU machine and then deployed on other commodity clusters or mobile devices. Caffe is written in clean, efficient C++ with CUDA used for GPU computation and Python and Matlab interface. Although it was developed on Linux, Caffe can run on most operating systems, such as Ubuntu, RHEL, CentOS, OS X and Windows.

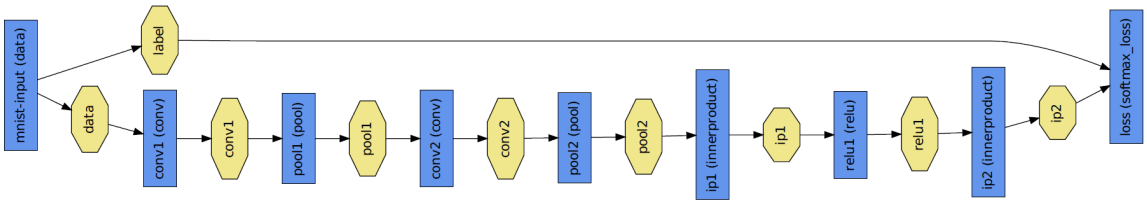

Figure 3.9: An MNIST digit classification example of Caffe network. Source[18]

Caffe stores and communication in 4-dimensional array called blobs [18]. A Caffe layer is the essence of a neural network layer. It takes one or more blobs as input and output one or more blobs to next layer. Layers have two key responsibilities: a forward pass that take inputs and produce the outputs, and a backward pass that execute BackPropagation algorithm, which is discussed above. Caffe also provides a complete set of layer types, including convolution, pooling, inner products, non-linearities (like ReLU and logistic), element-wise operation, and losses (like softmax and hinge). Figure 3.9 shows an example Caffe network used to classify MNIST digit. The yellow octagons are the data blobs produced or fed into layers. The blue rectangles are different layers. There is one input layer at the left and one output layer at the end. In the middle part, there are 7 hidden layers, including 2 convolution layers, 2 pooling layers, 2 inner product layers and 1 non-linearity layer (ReLU). In this figure, Backpropagation algorithm is not shown.

Chapter 4 System overview

4.1 Design

The whole system contains 3 main components: data capture, data process, and result display. The inputs of joints detection is a 2D color image, but to reconstruct the 3D model, I also need a depth image. It will be more efficient for reconstruction if human body mask is also provided. Therefore, I totally need 3 type images: color image, depth image, and body mask. Based on these, I choose Kinect as the input device. Then I also need a Windows PC and a working Caffe network on Windows. Figure 4.1 shows the workflow of the system.

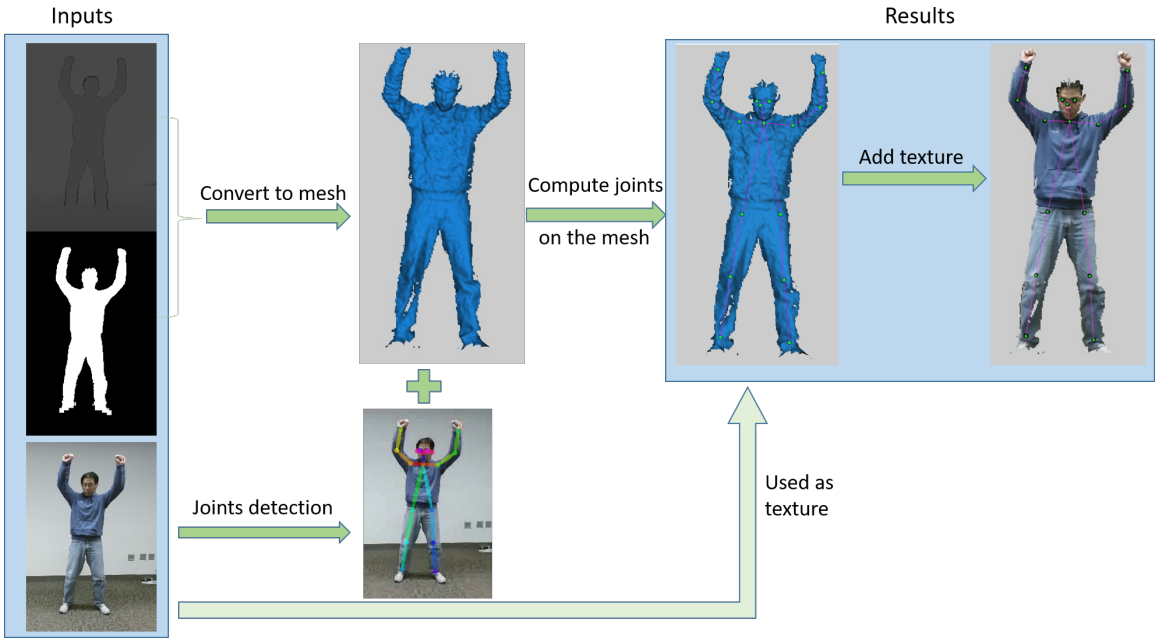

Figure 4.1: System workflow.

4.2 Setup

The hardware includes a Kinect mounted to a tripod and a PC machine. Figure 4.2 shows all the hardware that needed in the system. Because I need to stream image data from Kinect, so I deploy my code on a desktop running Windows 10. The desktop has i7-7700K @4.20Ghz, a 32GB RAM memory and the graphic card is GTX 1070 with GPU capability 6.1. To run Caffe, the graphic card at least has GPU capability 3.0. The operating system is 64-bits Windows 10. With this configuration, the system can process 3 4 frames per second. The bottleneck is the joints detection part. If there are more than one GPU device or more powerful GPU, the performance can be better.

Figure 4.2: System setup.

By using Kinect sensors, RGB color images, depth images and mask images can be obtained. Instead of using raw color image, which is 1920x1080, I preprocess the depth and color images, so that for each frame, color image and depth image are pixel-wise corresponded. Then the processed color image, which is 512x424, will be used to detect joints by using deep learning. The deep learning part can provide all the joints information on the 2D color image. Since depth image is coordinated with color image, so the joints should also be the same position in the depth image. Both the depth images and joints can be transformed to points cloud in 3D space by divided by the intrinsic matrix. Once get the 3D points cloud, meshes can be produced by triangulation. Once all the information are ready and will be rendered by OpenGL. I will discuss more details about the three components in next chapter.

Chapter 5 System implementation

5.1 Data buffer

This whole system is a multi-thread program, so an important thing is to deal with buffers. Because efficient data transfer can improve the system performance. There are four buffers among main components. The first one is used to cache data from Kinect. Because Kinect stream images faster than the processing procedure, there are many frames will be cached in the buffer.I implemented two kinds of buffer for this part, as shown in Figure 5.1. The first one is for live demo. It can cache constant number of frames, for example (10 frames in the thesis). Since the Kinect can provide more than 10 frames every second, some frames will be dropped, but it can track user's latest pose in 2 seconds. The second one is no limit number of frame and is used for no-drop frames. Because it will cache all the frames since the program start, the buffer will increase with time. This buffer will cost a lot of memory and it will crash once there is no more memory to allocate. I also try another solution. Since the bottleneck is the joints detection and it can only process 3 4 frames every second. So I can write the streaming image to the disk drive and read it into the memory when needed. It will not use too much memory, but need large disk space, moreover, the extra write and read process will decrease the whole performance. The second buffer is cache the output of joints detection. In practical, the OpenGL rendering procedure can process faster than the joints detection, so there I also just set the frame buffer number as 10, the same as the first buffer. The third and fourth buffer are used to store depth image and mask image respectively. These two images are used for 3D body model reconstruction.

Figure 5.1: System buffer.

I also setup several flags to control dataflow. READ_FLAG is to decide whether should capture data from Kinect or not; PROCESSED_FLAG shows joints detection components can read data from buffer or not; RENDER_FLAG to indicates is there any result to be display or not. Those flags help control the system work flow.

5.2 Compile Caffe on Windows

Although it's easy to setup Caffe on Linux, I have to setup it on Windows with Visual studio 2013, because I need Kinect to capture images. There are several third

party dependencies that needed by Caffe, such OpenCV, CUDA, Boost, OpenBLAS, GFlags, Glog, ProtoBuf, and LevelDB. I tried very long time to compile Caffe on Windows, and finally follow Neil's blog[6], I setup Caffe successfully on Windows. I use OpenCV2.4.11, not OpenCV3, because the joints detection part only works with OpenCV2. The latest CUDA8.0 at that moment. Because the difference between Linux system library and Window's library, there are a lot of compiling errors during this procedure, like method *close*() in Linux is corresponding to *_close*() in Windows.

5.3 Data streaming

Kinect can provide 6 data sources: ColorFrameSource, DepthFrameSource, BodyFrameSource, InfraredFrameSource, BodyIndexFrameSource, AudioSource. I only need first 3 data source. To get the 3 type image data, should follow this procedure "Sensor – Source – Reader – Frame – Data". The Sensor only need to open once, but the other steps are need to be went through for every data source. Based on an example in the Kinect SDK V2.0, naming "Coordinate Mapping Basics-D2D", I can get pretty clean mask images and color images which are pixel-wise coordinated with depth images. After the coordinating operation, the color image will be resized from 1920x1080 to 512x424. Although Kinect SDK provide a method ($MapDepthFrameToColorSpace()$) can do that, if the depth image and the color image are not aligned well, we can calibrate the depth camera and color camera respectively and get two RT matrices. Based on the two RT matrices, for each pixel on the depth image, we can find corresponding pixel on the color image. That's how we implement the mapping function by our own.

Smaller color image means more images can be loaded to GPU, so the performance will be better. On the other hand, joints detected on the color images will be in the same location on the depth images. Figure 5.2 show an example. The left column are the preprocessed color image and its corresponding depth image. The right column shows that joints detected on the color image will be at the same location of depth image.

5.4 Detect joints on 2D color image

Zhe Cao *et al.* present an efficient approach to detect the 2D pose in an image[9]. I use their network and trained model as in [9]. Their system is originally developed on Linux and only for 2D data, but I transport the codes to Windows 10 and tracking body in 3D space.

Human 2D pose estimation is the problem of finding anatomical keypoints or joints. A common top-down way [31, 12, 27] is to apply a person detector first and then perform person pose estimation. But if the person detector fails, there is no way to do the pose estimation. Zhe Cao *et al.* [9] present an efficient method for pose estimation with state-of-the-art accuracy on multiple public benchmarks. It's a bottom-up method and support multi-person pose estimation. Unlike the top-down methods whose runtime is proportional to the number of people in the image, Zhe's

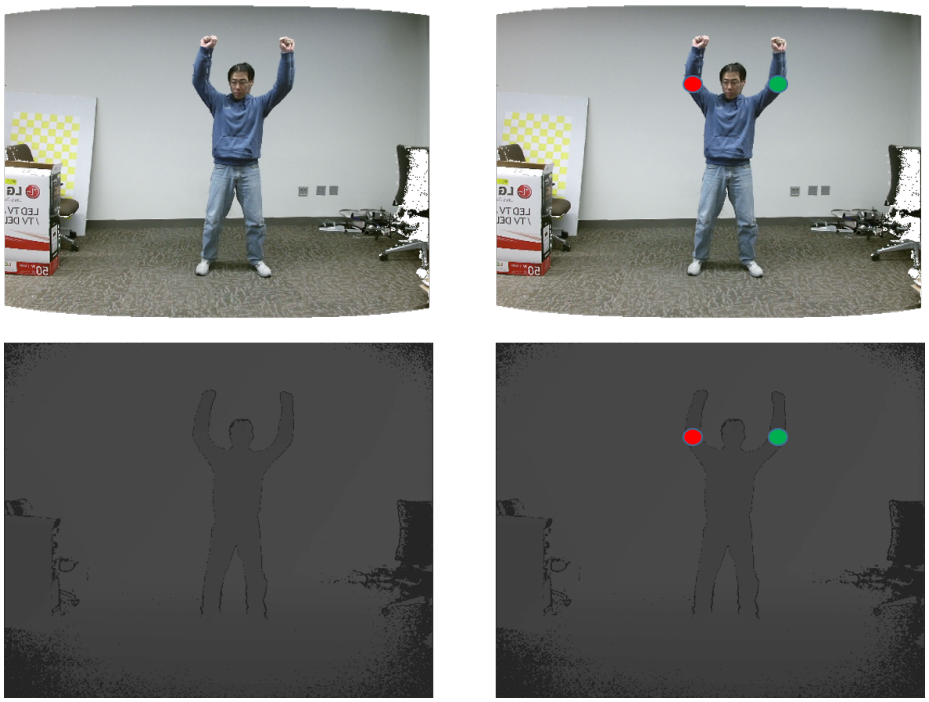

Figure 5.2: Color image is pixel-wise corresponded to depth image.

method can decouple runtime complexity from the number of people. I didn't handle multiple person right now, but I do need their high accuracy in joints detection.

Figure 5.3 shows essential part of the architecture of the deep learning network, which is a two branch multi-stage CNN. There are different numbers and different size convolution layers in each stage. The input of the first stage of each branch is a set of feature maps F, which can be extracted by applying convolution, non-linearity and pooling. Each input image will be analyzed by a convolutional network to generate a set of feature maps F. The convolutional network is initialized by the first 10 layers of VGG-19 [30]. Figure 5.4 is the actual architecture of the system. There are 10 convolution layers, 10 ReLU layers and 3 pooling layers before the two-branch multi-stage CNN. In this project, there are 6 stage two-branch CNN.

About the two branches, one is for joints detection and the other one is for joints association. I only need the joints detection part and associate joints by predefined relationship. Although the joints are detected on the color image, since the depth image is pixel-wise corresponded with color image, as shown in figure 5.2. So all the detected joints will also be known on the depth image.

5.5 Convert 2D depth to 3D

After the joints detection, joints positions will also be known on the depth image. I use the built-in camera intrinsic parameters to generate 3D points cloud. The joints

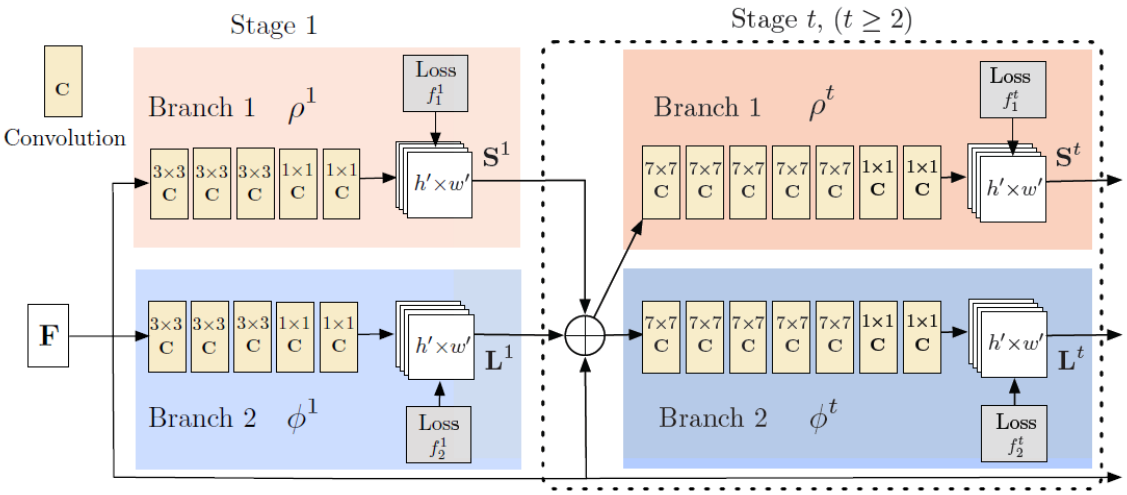

Figure 5.3: Architecture of the two-branch multi-stage CNN. Source[9]

will also be converted to 3D space, called *depth2pointmap*().

$$KK = \begin{bmatrix} 365.5953 & 0.0 & 260.1922 \\ 0.0 & 365.5953 & 209.5835 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}$$

To make the conversion more efficient, mask images are used in this procedure. Because a whole depth map also includes background, not just human body. So with the mask, I can only convert depth map which is inside the mask to points cloud. Kinect produce mask image based on depth threshold.

Once have all the 3D points, it's easy to generate 3D mesh by connect every 3 closest points, called *pointmap2mesh*(). To make sure there is no noise points, an extra operation (*RemoveLongFaces*()) is needed. Because some noise points are far away from the main point clouds, but during the *pointmap2mesh*(), the noise points will also be connected to other points, so these faces are long faces and noise faces, which should be removed. After the *RemoveLongFaces*(), a clean mesh is ready to render out.

5.6 2D joints to 3D joints

Although detected 2D joints can also be projected to 3D space using the same depth value as the same (x, y) location on the depth map. To make sure all the joints can be rendered on the mesh, I also did some extra work. Although the detected joints are in the 3D space, but maybe they are not exactly on the produced mesh. To make sure that, a ray is shot through a joint and should intersect with the mesh, the intersection is the new location of that joint. But there may be some depth value missing on the depth map , which means there may be some holes on the produced mesh. In that case, the ray will not intersect with the mesh. If that happens, I choose use the original detected joints location.

5.7 Render 3D mesh with OpenGL

OpenGL is short for Open Graphics Library, and is a cross-platform application programming interface (API) for rendering 2D and 3D graphics. *Freeglut* is a third party library that implements all the OpenGL interfaces and it supports for 64-bits application program. To use OpenGL, first need to initialize the OpenGL environment by calling $glutInit()$, then need create a OpenGL window to show the rendered results by $glutCreateWindow()$. After setting the display function handler by $glutDisplayFunc()$, $glutMainLoop()$ need to be called to start rendering. Since the mainLoop is a infinity loop and the display function will be called every iteration.

The color image will also be used in this procedure as a rendering texture. After adding texture to the 3D mesh, the 3D human body model will be more realistic. It can more clearly show how a person move and can track the person's body in 3D.

There is one thread keep executing the OpenGL functions. So the infinity loop doesn't affect reading data from Kinect and process image. This is the advantage of using multi-thread.

5.8 Multi-threads

A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler[32]. In most cases, a thread is a component of a process and multiple threads can exist within one process, executing concurrently. Multiple threads will share resources within the same process. The usage of multiple threads can improve the program performance. In C/C++, there is a third party library *pthread* that works both on Linux and Windows. It's very easy to create a new thread by calling $pthread\_create()$.

In the system, one thread only capture image data from Kinect and push data into 3 image buffers (color image, depth image and mask image). The second thread convert color image data from $CV :: Mat$ to a customized data structure. The third thread read color image from the data structure and detect joints location and push joints information to a joint buffer. The fourth thread read joints information from joint buffer, depth image from depth buffer and mask image from mask buffer. Then produce a 3D human body mesh with detected joints and render it to the screen.

Figure 5.4: The whole architecture of the deep learning network.

Chapter 6 Results

The detected joints are shown in green spots and the red lines are the skelton. Figure 6.1 and Figure 6.2 show some results with and without texture. As shown in these two figures, the joints detection are very accurate, more clearly when shown without texture. The system is robust to extreme poses. During the user turning around, the system can still track the key joints very accurately, while the Kinect usually fail when the user turns a little bit. Moreover, the joints number are different depend on the person is frontal face or not, but Kinect cannot separate. So the system can also decide whether the user is looking at the camera or turning back to the camera without face detection. The demo video can be found at `https://www.youtube.com/embed/VxLqbrKHH-g`

Figure 6.1: Results with texture.

Zhe Cao *et al.* did some joints detection experiments on two benchmarks: (1) the MPII human multi-person dataset [1] and (2) the COCO 2016 keypoints challenge dataset [23]. The following table is from [9] and shows the comparison results. As it shows, this deep learning network can get more accurate joints detection result than the other two state-of-art methods in a very short time.

Figure 6.2: Results without texture.

Method	Head	Sho	Elb	Wri	Hip	Kne	Ank	mAP	s/image
DeeperCut [13]	78.4	72.5	60.2	51.0	57.2	52.0	45.4	59.5 485	
Iqbal *et al.* [15]	58.4	53.9	44.5	35.0	42.2	36.7	31.1	43.1	10
Zhe Cao *et al.* [9]	91.2	87.6	77.7	66.8	75.4	68.9	61.7	75.6	0.005

6.1 Compare with Kinect

Figure 6.3 shows the comparison result. In Figure 6.3, Kinect cannot accurately detect joints like the shoulder, hips and knees. Actually in most cases, Kinect cannot detect the knees very accurately. Figure 6.4 shows more results and can prove that. Joints that are not accurately detected are circled out in Figure 6.4. It is clearly that ankles are even detected outside of human body when the user jump up. When the user turn around some angle, some joints will not be detected. The second row of Figure 6.4 shows the joints detection result for the same pose. Compared to Kinect, my system can detect joints much more accurate even extreme pose.

6.2 Runtime analysis

Although my whole system take advantage of multi-thread and there are total 4 threads in this system: one is for reading data from Kinect; the second one is for convert the Kinect image data to the network input; the third one is for running network to detect joints; the last one is running OpenGL to render results. The

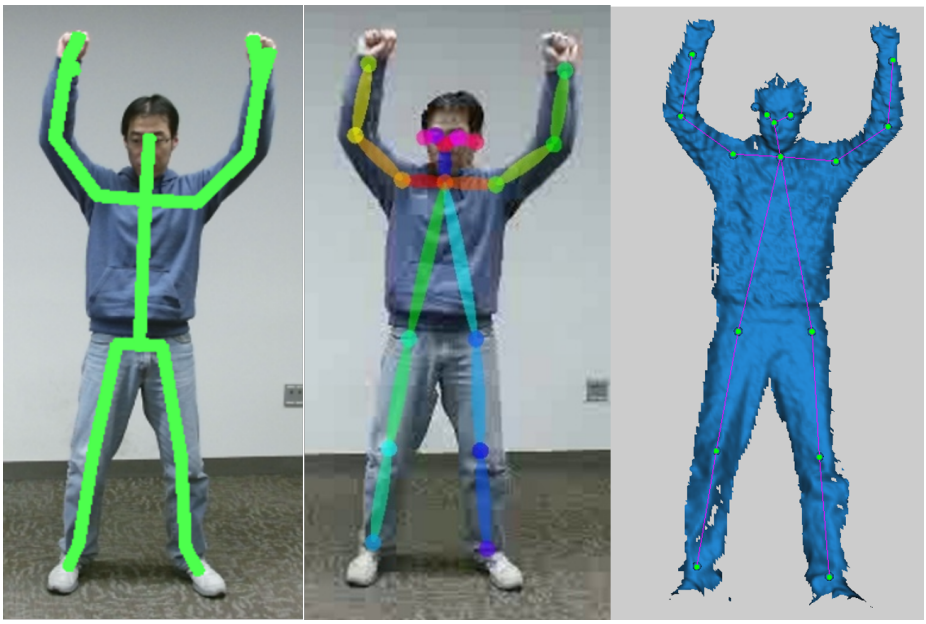

Figure 6.3: Left column is the Kinect result, the middle column is joints detected by my system, the right column is the 3D mesh

system can process 3 4 frames every second. The rendering procedure is realtime, so the bottle neck is the joints detection. I run the joints detection part separately on Linux and it can process 6 frames per second on the same desktop. The Windows operating system should occupy some GPU resources, which can account for the performance difference. Since the network support multiple GPU, so if there are more than one GPU devices, the performance can be better.

Figure 6.4: Joints detect result. First row are the Kinect outputs. Second row are my results.

Chapter 7 Conclusion

In this thesis, I set up a 3D body tracking system, which take advantage of deep learning and multi-thread programming. The system use Kinect as input device, detect body joints by applying the deep learning network released in [9] on a color image which is pixel wise coordinated with a depth map and convert the 2D depth map to a 3D human model with joints and 3D skeleton. Although Kinect can track human body in real time, but it cannot detect joints very accurately and cannot tell whether the user is facing to the camera or not. According to the experiments results, the system can detect joints more accurate than Kinect and can show 3D result. Although the joints detection is accurate, it is still the bottleneck of the system. The whole system can process 3 4 frames per second. If there are more GPU resources (more GPU devices or more powerful GPU), the performance can be better.

Bibliography

[1] Mykhaylo Andriluka et al. "2d human pose estimation: New benchmark and state of the art analysis". In: *Proceedings of the IEEE Conference on computer Vision and Pattern Recognition*. 2014, pp. 3686–3693.

[2] *Backpropagation*. URL: https://en.wikipedia.org/wiki/Backpropagation.

[3] Pouya Bagherpour, Seyed Ali Cheraghi, and Musa bin Mohd Mokji. "Upper body tracking using KLT and Kalman filter". In: *Procedia Computer Science* 13 (2012), pp. 185–191.

[4] V. Belagiannis and A. Zisserman. "Recurrent Human Pose Estimation". In: *2017 12th IEEE International Conference on Automatic Face Gesture Recognition (FG 2017)*. 2017, pp. 468–475. DOI: 10.1109/FG.2017.64.

[5] Vasileios Belagiannis and Andrew Zisserman. "Recurrent human pose estimation". In: *Automatic Face & Gesture Recognition (FG 2017), 2017 12th IEEE International Conference on*. IEEE. 2017, pp. 468–475.

[6] *Build Caffe in Windows with Visual Studio*. URL: https://initialneil.wordpress.com/2015/01/11/build-caffe-in-windows-with-visual-studio-2013-cuda-6-5-opencv-2-4-9/.

[7] Adrian Bulat and Georgios Tzimiropoulos. "Human pose estimation via convolutional part heatmap regression". In: *European Conference on Computer Vision*. Springer. 2016, pp. 717–732.

[8] BVLC. *Caffe*. 2014. URL: http://caffe.berkeleyvision.org/.

[9] Zhe Cao et al. "Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields". In: *arXiv preprint arXiv:1611.08050* (2016).

[10] Xianjie Chen and Alan L Yuille. "Articulated pose estimation by a graphical model with image dependent pairwise relations". In: *Advances in Neural Information Processing Systems*. 2014, pp. 1736–1744.

[11] K. M. G. Cheung, S. Baker, and T. Kanade. "Shape-from-silhouette of articulated objects and its use for human body kinematics estimation and motion capture". In: *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.* Vol. 1. 2003, I–77–I–84 vol.1. DOI: 10.1109/CVPR.2003.1211340.

[12] Georgia Gkioxari et al. "Using k-Poselets for Detecting People and Localizing Their Keypoints". In: *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*. CVPR '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 3582–3589. ISBN: 978-1-4799-5118-5. DOI: 10.1109/CVPR.2014.458. URL: http://dx.doi.org/10.1109/CVPR.2014.458.

[13] Eldar Insafutdinov et al. "DeeperCut: A Deeper, Stronger, and Faster Multi-person Pose Estimation Model". In: *Computer Vision – ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part VI*. Ed. by Bastian Leibe et al. Cham: Springer International Publishing, 2016, pp. 34–50. ISBN: 978-3-319-46466-4. DOI: 10.1007/978-3-319-46466-4_3. URL: https://doi.org/10.1007/978-3-319-46466-4_3.

[14] Intel. *Real sense*. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/realsense-overview.html/.

[15] Umar Iqbal and Juergen Gall. "Multi-person Pose Estimation with Local Joint-to-Person Associations". In: *Computer Vision – ECCV 2016 Workshops: Amsterdam, The Netherlands, October 8-10 and 15-16, 2016, Proceedings, Part II*. Ed. by Gang Hua and Hervé Jégou. Cham: Springer International Publishing, 2016, pp. 627–642. ISBN: 978-3-319-48881-3. DOI: 10.1007/978-3-319-48881-3_44. URL: https://doi.org/10.1007/978-3-319-48881-3_44.

[16] Arjun Jain et al. "Learning human pose estimation features with convolutional networks". In: *arXiv preprint arXiv:1312.7302* (2013).

[17] Dae-Sik Jang, Seok-Woo Jang, and Hyung-Il Choi. "2D human body tracking with structural Kalman filter". In: *Pattern Recognition* 35.10 (2002), pp. 2041–2049.

[18] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv preprint arXiv:1408.5093* (2014).

[19] Ujjwal Karn. *A Quick Introduction to Neural Networks*. 2016. URL: https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks.

[20] Ujjwal Karn. *An Intuitive Explanation of Convolutional Neural Networks*. 2016. URL: https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets.

[21] R. Kehl, M. Bray, and L. Van Gool. "Full body tracking from multiple views using stochastic sampling". In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 2. 2005, 129–136 vol. 2. DOI: 10.1109/CVPR.2005.165.

[22] Win Kong, Aini Hussain, and Mohd Hanif Md Saad. "Essential human body points tracking using kalman filter". In: *Proceedings of the World Congress on Engineering and Computer Science*. Vol. 1. 2013, pp. 503–507.

[23] Tsung-Yi Lin et al. "Microsoft coco: Common objects in context". In: *European conference on computer vision*. Springer. 2014, pp. 740–755.

[24] Alejandro Newell, Kaiyu Yang, and Jia Deng. "Stacked hourglass networks for human pose estimation". In: *European Conference on Computer Vision*. Springer. 2016, pp. 483–499.

[25] *optitrack*. URL: http://optitrack.com/.

[26] Tomas Pfister, James Charles, and Andrew Zisserman. "Flowing convnets for human pose estimation in videos". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2015, pp. 1913–1921.

[27] L. Pishchulin et al. "Articulated people detection and pose estimation: Reshaping the future". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. 2012, pp. 3178–3185. DOI: 10.1109/CVPR.2012.6248052.

[28] Javier Romero, Matthew Loper, and Michael J. Black. "FlowCap: 2D Human Pose from Optical Flow". In: *GCPR*. 2015.

[29] Jamie Shotton et al. "Real-Time Human Pose Recognition in Parts from a Single Depth Image". In: 2011.

[30] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[31] Min Sun and Silvio Savarese. "Articulated part-based model for joint object detection and pose estimation". In: *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE. 2011, pp. 723–730.

[32] *Thread (computing)*. URL: https://en.wikipedia.org/wiki/Thread_(computing).

[33] Jonathan J Tompson et al. "Joint Training of a Convolutional Network and a Graphical Model for Human Pose Estimation". In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani et al. Curran Associates, Inc., 2014, pp. 1799–1807. URL: http://papers.nips.cc/paper/5573-joint-training-of-a-convolutional-network-and-a-graphical-model-for-human-pose-estimation.pdf.

[34] Stanford University. *CS231n Convolutional Neural Networks for Visual Recognition*. URL: http://cs231n.github.io/convolutional-networks.

[35] Shih-En Wei et al. "Convolutional pose machines". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 4724–4732.

[36] Mao Ye et al. "Accurate 3d pose estimation from a single depth image". In: *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE. 2011, pp. 731–738.

Vita

I am Qingguo Xu and I was born in 1988, Zoucheng, China. I got my bachelor degree in Computer Science department of Xi'an Jiao Tong University in 2011. I came to University of Kentucky in 2013. I won the Kentucky Opportunity Fellowship in 2015.