Ruiying Yang

APPLICATION OF SINGLE HUMAN POSE ESTIMATION: GENERATING REAL-TIME CHARACTER ANIMATION

Faculty of Information Technology and Communication Sciences (ITC)
Master's thesis
April 2022

Abstract

Ruiying Yang: Application of Single Human Pose Estimation: Generating Real-time Character Animation
Master's thesis
Tampere University
Master's Degree Programme in Data Engineering and Machine learning
April 2022

The capture of human pose has been deployed in various applications, such as automatically generating 2D animations from human pose and healthcare applications. The human pose can be predicted by the deep neural network, thus different architectures of neural network were proposed. However, the proposed human pose estimation models were trained and tested on the different datasets, through which, the performances of different architectures cannot be evaluated on a common standard. Therefore, this thesis trained and tested five architectures for human pose estimation on the same dataset (Human3.6M), compressing stacked Hourglass, HRFormer, LiteHRNet, MobileNetV2 and MobileHumanPose model and qualitative evaluation results show that stacked hourglass model can produce the most accurate predicted keypoints among these models, while the LiteHRNet is fastest for inference. Addition to testing prediction errors of different architectures on a determined dataset, this thesis built an application in Unreal Engine 4 (UE4) game engine to generate character's animations in real time based on the predicted 3D human keypoints, which are acquired from applying human pose estimation models for continuous frames, captured by WebCam. The generated real-time animations by using different models haven't shown the visible differences, but the depth values (z-axis) of predicted keypoints are not realistic compared to the human actions, which can be shown from the unexpected animations when human pose is rotating or moving in z-axis.

Keywords: human pose estimation, deep neural network, animation

Contents

List of Figures

List of Tables

1 Introduction

Generally, in most software, like games, the character animations are manually crafted by artists, which is predefined assets. The motion capture is a common technique, utilizing specific equipment and software to capture the motions and actions of human and the motion data can be used for making animation in games or for other purposes. However, the equipment used in motion capture is expensive and operating it requires professional training, thus the most of artists design and create animations manually. If character performs different actions and movements, artists need to make different animations. For games, providing various animations increases playability, but making huge number of animations, especially very realistic animations leads to heavy workloads. Moreover, for an interactive application, a real-time actions of human needs to be captured, for example, in games, players can use own actions to control character's animations and real-time actions of human can be used in self-driven filed, like detecting actions of pedestrian.

Therefore, the main objective of this thesis is creating an application in Unreal Engine 4 (UE4) game engine where the character is animated based on the player's actions by utilizing 3D human pose estimation (HPE), involving training and testing 3D single pose estimation models using different neural network architectures to generate 3D locations of skeletal joints from an RGB image and animating character in real-time by calculating rotations of character skeletal joints, which utilizes inverse kinematics(IK) provided by UNREAL animation system.

HPE is estimating the configuration of human body parts (like locations of skeletal joints, body shapes) from images or videos. This thesis only built the models for estimating 3D locations of humane skeletal joints, which is also called pose landmark's estimation.

The accuracy of 240 research papers from 2014 to 2020 were analyzed and compared in a survey (Zheng et al. 2020), based on that, 2D to 3D lifting approaches outperform direct estimation approaches in general, which is because 2D HPE with 2D pose annotations is easily achievable and high performance has been reached for a single person while acquiring accurate 3D pose annotations for 3D HPE is more difficult than its 2D counterpart.

The evaluation results of 240 research papers provided by the survey (Zheng et al. 2020) show that the models, Stacked Hourglass (Yuan et al. 2021), HRFormer (Yu et al. 2021), Lite-HRNet (Sandler et al. 2018), MobileNetV2 (Newell, Yang, and Deng 2016), have relatively higher accuracy in 2D HPE. And the model, Mobile-HumanPose has relatively better performance in 3D HPE (Sangbum Choi, Seokeon Choi, and Kim 2021). However, the performances provided by these research works were evaluated by using different test datasets and different evaluation metrics. Therefore, this thesis tested several relatively better models mentioned in this survey using the same test dataset and evaluation metrics. Moreover, the speed of model is very important in the real-time applications, thus this thesis also evaluated the inference speed by using different pose models.

The existing 2D pose models produce 2D landmark positions while the 3D positions are needed for animating characters in the 3D world (Yuan et al. 2021; Yu et al. 2021; Sandler et al. 2018; Newell, Yang, and Deng 2016), so the 2D to 3D lifting method was applied following the 2D HPE models (Martinez et al. 2017).

The structure of this thesis is composed of five parts. Chapter 2 introduces the necessary concepts and theories applied in the human pose estimation, including deep neural network, inverse kinematics, etc. Chapter 3 explains the advanced architectures of deep neural networks for 2D pose estimation and 3D pose estimation. Chapter 4 provides experimental training and evaluation of different human pose estimation models from a single RGB image and in a real-time application. Chapter 5 makes a conclusion of this thesis project.

2 Theory

In recent researches, the neural network model is popular for human pose estimation (HPE) due to its excellent prediction accuracy and most of neural network models for HPE are based on convolutional neural networks (CNN), which is very good in image processing.

Generally, neural network is an artificial intelligence method, used for training computers to process data and make prediction from the input data. Inspired by the human brain, the neural network model consists of the artificial neurons that form a complex and interconnected network to simulate human brain. The first artificial neuron model is called M-P Neuron that process the input data by multiplying it with its corresponding weights. However, the weights of M-P neuron is determined in advance and in order to adjust weights based on the input data and output data, the perceptron model is proposed. Multiple perceptrons constitute a perceptron layer and multiple perceptron layers constitute the neural network.

In state-of-art neural network architectures, different layers are designed to process data on different purposes, like convolutional layer, and different neural networks are used for different tasks. Thus the details of layers in neural networks and weight computations are introduced in Chapter 2 and Chapter 3 describes the several neural network architectures used for HPE. Additionally, writing the computer program to build and train a neural network is complex, but it's convenient to use existing frameworks, like TensorFlow and PyTorch. This thesis tests the neural network model in the real-time, thus it's good to use TensorFlowLite for fast inference. Therefore, after training with PyTorch, the conversion from PyTorch model to TensorFlowLite model is required and this conversion needs to use ONNX format as an intermediary. These neural network frameworks are introduced in Chapter 2 as well. In addition to neural network model, generating character animations needs IK techniques, by which rotations of each character joint are acquired by defining locations of joints.

In this section, the basic knowledge of neural network, Convolutional Neural Networks (CNN), some deep learning frameworks and Unreal Engine 4 (UE4) game engine are introduced.

2.1 Supervised and unsupervised learning

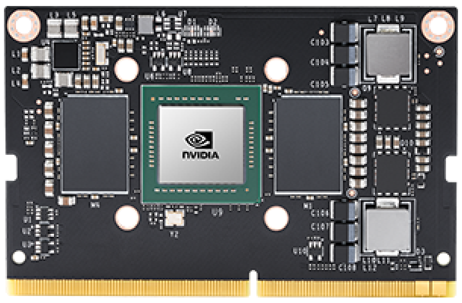
Machine learning (ML) is a relatively recent field of artificial intelligence study (AI). In general, machine learning (ML) learns from data, which means it detects the regulation in the data and predicts the outcomes for previously unknown data based on that regulation. There are two types of learning: supervised and unsupervised. Both learning methods find features, which are measurable data retrieved from datasets using a specific extraction approach. A model gets features from the input data and the expected output in supervised learning. During training, the model is fine-tuned until it can accurately predict unknown data. Unsupervised learning, on the other hand, does not require the intended output and instead extracts features from the data. Computers or robots are trained to perform natural language processing (NLP), image recognition, and other tasks.

2.2 Deep learning and its applications

Deep learning (DL) is the neural network with multiple layers in general. Its origin is traced back to the perceptron and the Boltzmann machine. Multilayer Perceptron (MLP), which incorporates many layers of perceptrons, is the structure of perceptron-based deep learning. Because it trains the neural network using predicted output, it is classified as supervised learning. For Boltzmann Machine-based deep learning (RBM), deep Boltzmann Machine and Deep Belief Network (DBN) are derived from numerous Restricted Boltzmann Machines. They are examples of unsupervised learning.

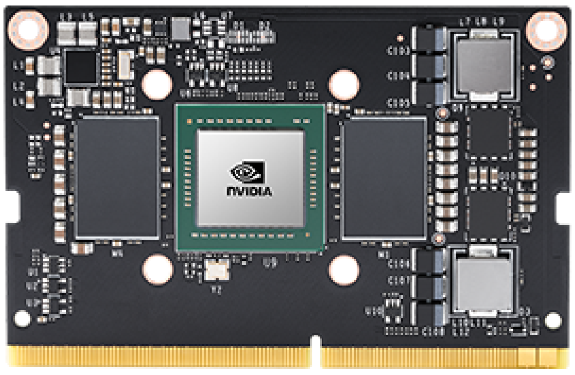
DL is aided by the advancement of the Internet, communication technologies, and hardware. On the Internet, people can quickly download massive public databases of images, audios, and other media. Hardware support for DL is provided by the GPU's high performance. The Graphics Processing Unit (GPU) is responsible for rendering graphics for display on electronic devices. Because it incorporates multiple processing units, it excels at parallel computing. As the Figure 2.1 shows, NVIDIA created a series of GPUs, such as the GeForce, Jetson, and Titan RTX for gaming, embedded systems, and scientific computation respectively. NVIDIA's CUDA Toolkit, which provides a development environment for constructing high-performance GPU-accelerated applications, is created to assist parallel computing (NVIDIA 2022a). CUDA reduces the time it takes for a GPU to complete a task. NVIDIA also built and developed cuDNN, a a GPU-accelerated library of primitives for deep neural networks (NVIDIA 2022b), in order to speed up DL.

Speech recognition, image recognition, and games are just a few examples of how DL can be used. According to the research, traditional voice recognition algorithms use Gaussian Mixture Model (GMM) and Hidden Markov Model (HMM) (G. Hinton et al. 2012). However with the help of DL, the initial performance limit was breached. Traditional picture identification algorithms include the Scale-Invariant Feature Transform (SIFT), Fishier Vector (FV), and others. According to the research, the performance of image recognition has substantially increased (Krizhevsky, Sutskever, and Geoffrey E Hinton 2012). In games, DL assists the



(a) Geforce RTX 3090 Ti

(b) Titan RTX



(c) Jetson TX2 NX Module

***Figure** 2.1 NVIDIA GPU series (NVIDIA 2022c)*

computer in achieving high scores. For a game like StarCraft II, it is complicated because it has difficulties in multi-agent problem, imperfect information, a large action space, a large state space and delayed credit assignment. DeepMind conducts research and developed SC2LE, a learning environment for experimenting with deep reinforcement learning (Vinyals et al. 2017).

2.3 Neural Network

2.3.1 The brief history of Neural Network

Mcculloch and Pitts realized the mathematical model of a neural network, McCulloch-Pitts Neuron (M-P Neuron), of a neural network by modeling biological neuron in the 1940s (Mcculloch and Pitts 1943). The perceptron proposed was in 1958, which could have its weights calculated by a computer after training (Rosenblatt 1958). Minsky, on the other hand, discovered that the perceptron is incapable of learning the exclusive-or function. This problem was solved in the 1980s by developing the Back Propagation (BP) technique on MLP (D. E. Rumelhart, McClelland, and PDP Research Group 1986). The pooling layer, which was the structure of Convolutional Neural Networks, was added to the neural network in 1989 (LeCun et al. 1989).

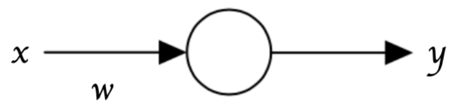
To train this network, they employed the BP algorithm. Nonetheless, the training is time-consuming, and overfitting was a problem. Since 2011, GPU advancements have boosted DL's performance in image recognition, speech recognition, and other areas. In addition, the growth of the Internet and communication technology has bolstered DL.

2.3.2 M-P Neuron

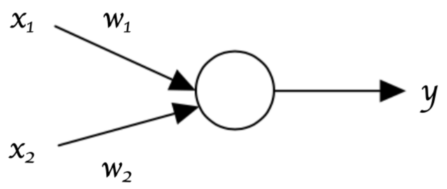
Mcculloch and Pitts proposed M-P Neuron (Mcculloch and Pitts 1943). As the Equation 2.1 shows, multiple input nodes, $\{x_i | i = 1, \ldots, n\}$, correspond to a single output node, y, in an M-P Neuron. Each input x_i is multiplied by its weight w_i, and the result is y. If the sum is more than the threshold h, y will be 1. Otherwise, y will be 0.

$$y = f\left(\sum_{i=1}^{n} w_i x_i - h\right) \tag{2.1}$$

The M-P Neuron's form is altered to perform logical operations, such as NOT, OR and AND. Here, humans decide on their own weight and threshold. As the following Figure 2.2 shows, M-P Neuron hassingle-input-single-output (SISO) and double-inputs-single-output (DOSO) model. The M-P Neuron's SISO model is used to accomplish the NOT operation. The outcome will be 1 if the input was 0. If the input is 1, on the other hand, the output will be 0.



(a) single-input-single-output M-P Neuron



(b) double-inputs-single-output M-P Neuron

***Figure 2.2** M-P Neurons*

Similarly, the OR and AND operations are performed using the M-P Neuron's DOSO model. The expected output is shown on the Table 2.1. OR is used if $w_1 = 1, w_2 = 2, h = 0.5$. In this way, $y = f(x_1 + x_2 - 0.5)$. AND is implemented if $w_1 = 1, w_2 = 2, h = 1.5$ is used. In this way, $y = f(x_1 + x_2 - 1.5)$.

2.3.3 Perceptron

One of the M-P neuron's flaws is that the weight and threshold must be determined before to the operation. During training, the perceptron automatically pick these

Table 2.1 *M-P Neuron's input and output of OR, AND*

input x_1	**input** x_2	**output of OR**	**output of AND**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

parameters based on the dataset (D. E. Rumelhart, McClelland, and PDP Research Group 1986). The dataset and predicted output should be confirmed prior to training. The error between the actual and expected output are then be corrected. This process is depicted by the Equation 2.2 and Equation 2.3, where r is the expected output. The learning rate is α, and it is used to control the speed with which errors are corrected. The training will not be stable if the α is too large. If it is too small, the rate of convergence will be slower.

$$w_i \leftarrow w_i + \alpha(r - y)x_i \tag{2.2}$$

$$h \leftarrow h - \alpha(r - y) \tag{2.3}$$

The value of the expected output and real output determines whether or not to change the weight and threshold. The weight and threshold will not change if they are the same. If they aren't, they should be. If the perceptron is inactive in this case, the threshold will be lowered. Meanwhile, $x_i = 1$'s weight will be increased. The threshold will be raised if the perceptron is overly activated. Meanwhile, $x_i = 1$'s weight will be reduced. On the algorithm 1, this updating procedure is depicted as a pseudocode. The final parameters are different because the parameters are initialized by random numbers in the beginning of training.

2.3.4 Multilayer Perceptron (MLP)

MLP stands for multiple perceptron layers in a neural network. In this network, the input propagates forward. As the following Figure 2.3 shows, MLP contains three kinds of layers in general: input layer, hidden layer, and output layer. Weights connect the hidden layers to the other layers. Weights are used to connect each unit of the input layer to the perceptrons in the hidden layers. The outputs produced by these perceptrons are sent to the output layer. This link is based on weights as well. Without backpropagation (BP), the weights between the input and hidden layers are set at random. The algorithm 1 fine-tunes them between the hidden and output layers.

Algorithm 1: A pseudocode of updating weight and threshold of perceptron

```
Data: $N$ training samples $x_i$, the expected output $r_i$
Result: ideal $w_i$
$w_i \leftarrow$ random value;
$h \leftarrow$ random value;
decide threshold of error $e_h$;
while $error = 0$ or $error < e_h$ do
    while $i \leq N$ do
        fed new $x_i$;
        calculate $y$;
        if $y! = r_i$ then
            if $y = 0$ and $r = 1$ then
                decrease $h$;
                increase $w_i(x_i = 1)$;
            end
            if $y = 1$ and $r = 0$ then
                increase $h$;
                decrease $w_i(x_i = 1)$;
            end
        else
        end
    end
end
```

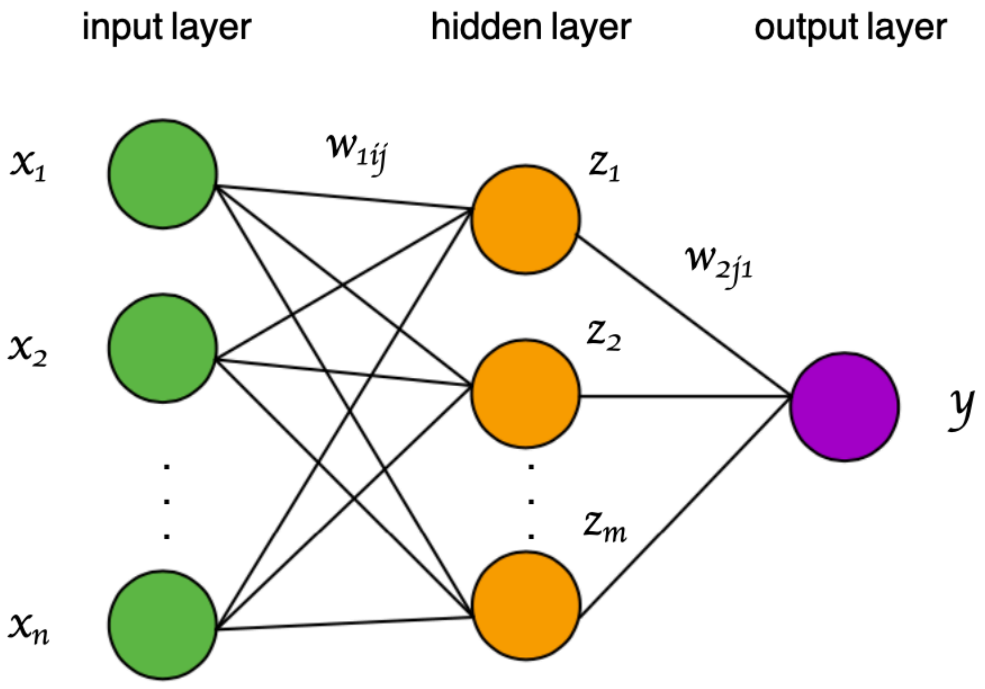


Figure 2.3 *Multilayer Perceptron with one output*

2.3.5 Loss function and activation function

Loss function measures the difference between the actual output r and expected output y. The Equation 2.10 is used in recursion. The loss function of bipartition is shown in the Equation 2.4.

$$E = -\sum_{n=1}^{N}(r_n ln y_n + (1 - r_n) ln(1 - y_n)) \tag{2.4}$$

The activation function inputs u, which is the sum of x_i and w_i (Equation 2.5) and transforms it into a linear or nonlinear form. The M-P neuron's activation function is the Step function (Equation 2.6). The output changes rapidly on $u = 0$ between 0 and 1. MLP used Sigmoid (Equation 2.7). As the following Figure 2.4 shows, the output of Sigmoid becomes gently compared to the Step function.

$$u = \sum_{i=1}^{n} w_i x_i \tag{2.5}$$

$$y = \begin{cases} 1, \text{if } u > 0 \\ 0, \text{if } u < 0 \end{cases} \tag{2.6}$$

$$f(u) = \frac{1}{1 + e^{-u}} \tag{2.7}$$

Another activation function is tanh, as the Equation 2.8 shows. The Figure 2.5 shows that its output ranges between -1 and 1.

$$tanh(u) = \frac{exp(u) - exp(-u)}{exp(u) + exp(-u)} \tag{2.8}$$

Another activation function is Rectified Linear Unit (ReLU), as the Equation 2.9 shows. Its output ranges between 0 and 1. The Figure 2.6 shows that if $u \leq 0$, the output will be 0. If $u \geq 0$, the output will be same as the input.

$$R(u) = max(u, 0) \tag{2.9}$$

2.3.6 Backpropagation (BP)

Backpropagation (BP) (David E. Rumelhart, Geoffrey E. Hinton, and Williams 1988) calculates the error by comparing the actual output r and expected output y. Then, the error is converted from the output layer to the previous layers to obtain the error for each layer. By fine-tuning weights in each layer using the gradient descent approach, the error is reduced. The error of the real and expected outputs,

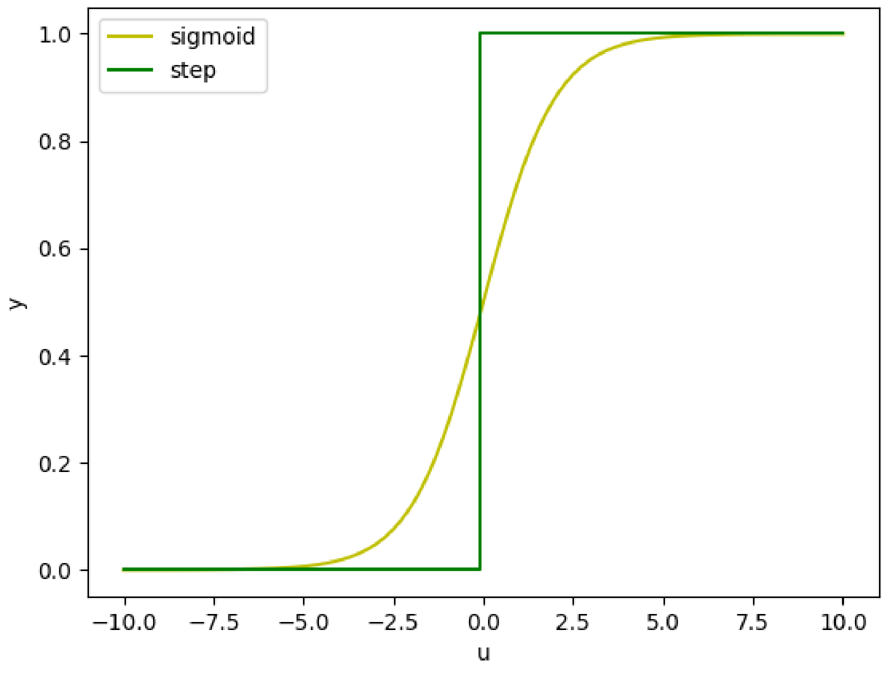


Figure 2.4 *Step and Sigmoid*

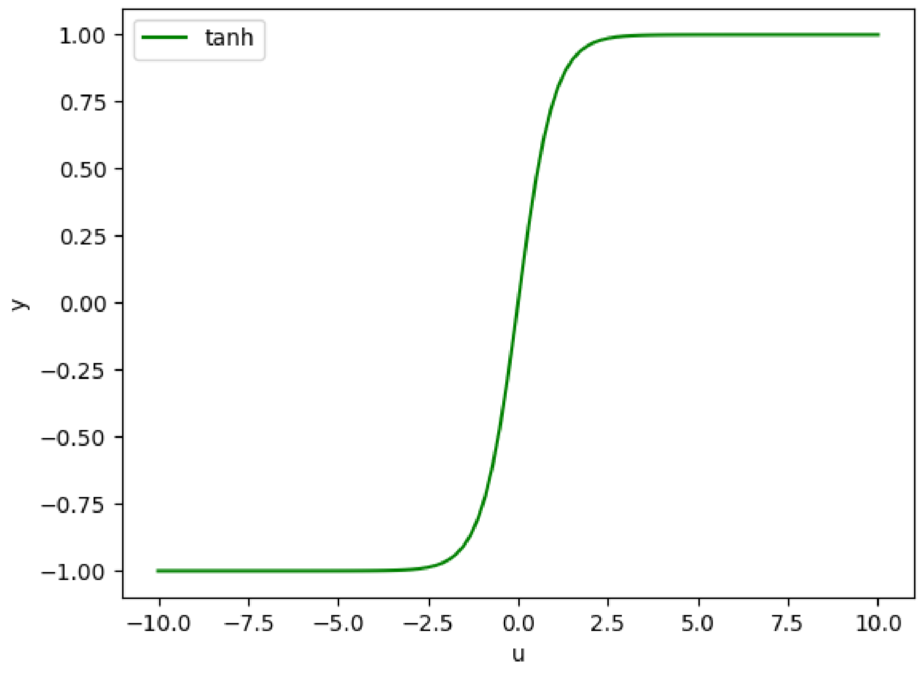


Figure 2.5 *tanh*

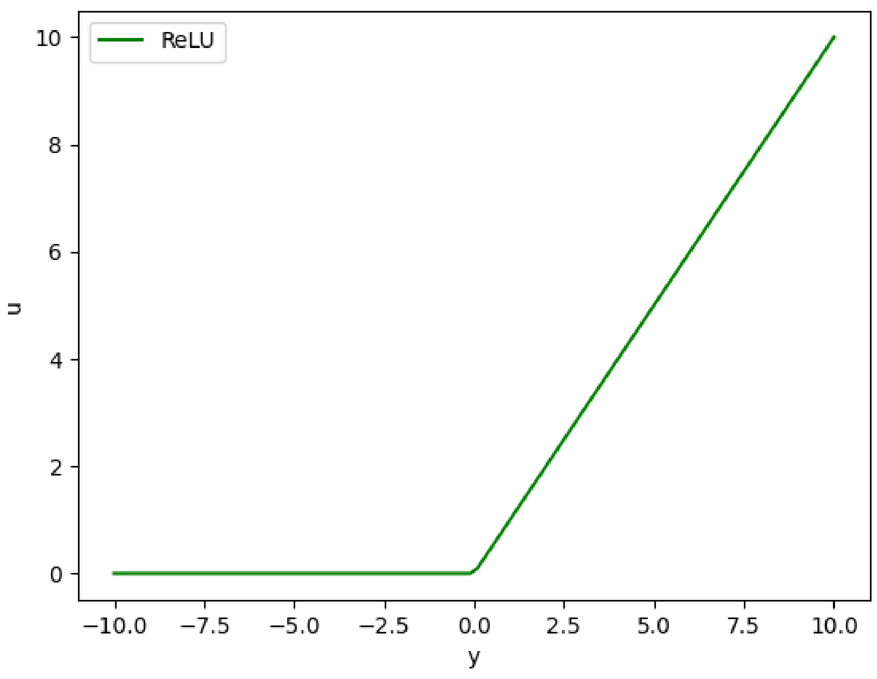


Figure 2.6 *ReLU*

as well as the gradient, are used to determine the new weight in this method. The inaccuracy is then reduced by fine-tuning the weights in each epoch until the best weight is obtained. As the following Figure 2.7 shows, the adjusted weight is the segment between w^0 and w^1. The new weight is decided given the adjusted weight until the optimum weight w^{opt} is obtained.

In the Gradient Descent Method (Lemaréchal 2010), the error is estimated using the Least Squares Error Function, Equation 2.10, which considers both the real and expected output. The discrepancy between the real and predicted outputs will be less if this error E approaches zero. In this way, MLP training is the process of fine-tuning the weights until the error approaches zero. The gradient is calculated using Equation 2.10's derivative. The adjusted weight will be raised if the inaccuracy is large, and vice versa. Equation 2.11 is used to get the adjusted weight. The learning rate is η, which is changed by the error.

$$E = \sum_{n=1}^{N} ||r_n - y_n|| \tag{2.10}$$

$$\Delta w = -\eta \frac{\partial E}{\partial w} \tag{2.11}$$

For a single perceptron, its error is calculated by Equation 2.12.

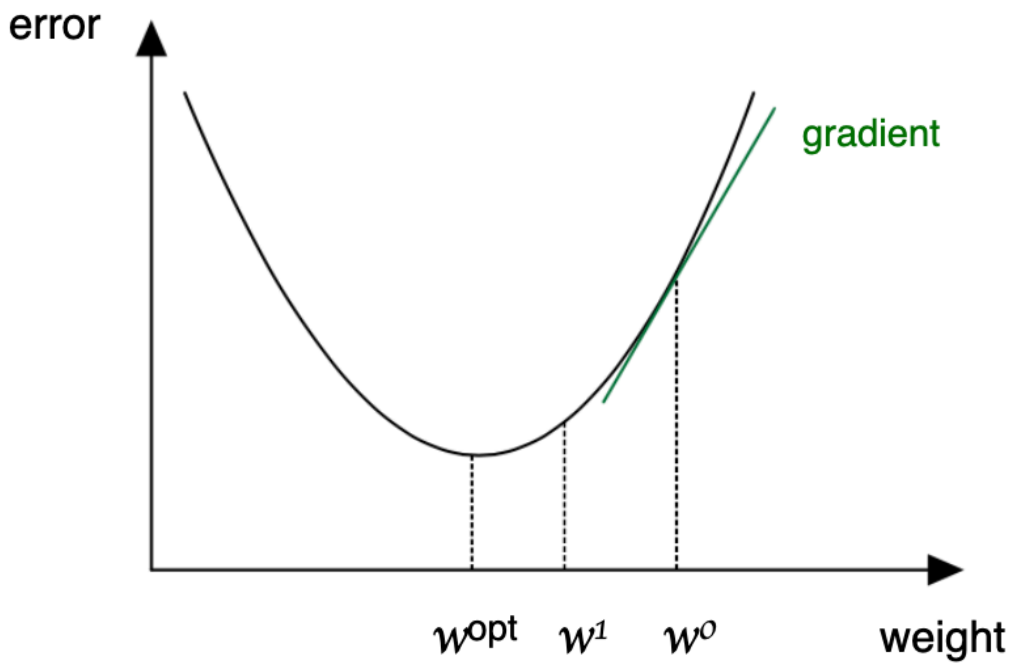


Figure 2.7 *Gradient Decsent*

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial w_i} \tag{2.12}$$

If the activation function is $f(u)$, assuming $y = f(u)$, the Equation 2.12 is calculated in this way, as the Equation 2.13 shows.

$$\begin{aligned} \frac{\partial E}{\partial w_i} &= -(r-y)\frac{\partial y}{\partial w_i} \\ &= -(r-y)\frac{\partial f(u)}{\partial w_i} \\ &= -(r-y)\frac{\partial f(u)}{\partial u}\frac{\partial u}{\partial w_i} \\ &= -(r-y)x_i\frac{\partial f(u)}{\partial u} \end{aligned} \tag{2.13}$$

If the activation function is Sigmoid, its derivative is shown in the Equation 2.14.

$$\frac{\partial f(u)}{\partial u} = f(u)(1 - f(u)) \tag{2.14}$$

Then, the Equation 2.14 is plugged into the result of Equation 2.13 and get the Equation 2.15.

$$\frac{\partial E}{\partial w_i} = -(r-y)x_i f(u)(1-f(u)) \tag{2.15}$$

Hence, given the $y = f(u)$, the adjusted weight is shown in the Equation 2.16

$$\Delta w_i = \eta(r-y)y(1-y)x_i \tag{2.16}$$

For a MLP, its error E is the sum of the error of each unit, as shown in the Equation 2.17

$$E = \frac{1}{2}\sum_{j=1}^{q}(r_j - y_j)^2 \tag{2.17}$$

If the MLP does not have hidden layers, as shown in the Figure 2.8. The derivative of its error is shown as the Equation 2.18, where w_{ij} denotes the weight between x_i and y_j.

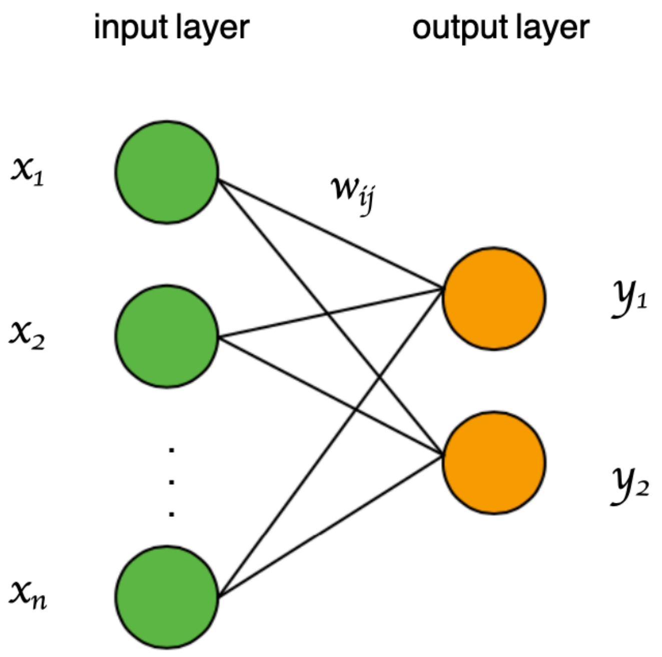


Figure 2.8 *Multilayer Perceptron without hidden layers*

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial w_{ij}} \tag{2.18}$$

The way of calculation is similar as the way of single perceptron, the adjusted weight is Equation 2.19.

$$\Delta w_{ij} = \eta(r_j - y_j)y_j(1-y_j)x_i \tag{2.19}$$

If a MLP has hidden layers with one output units, the figure is shown on the Figure 2.3. The adjusted weight between the input layer and hidden layer is shown as the Equation 2.20. The other weight between the hidden layer and output layer is shown as the Equation 2.21. In this equation, w_{1ij} denotes the weight between the input layer and the hidden layer and w_{2j1} denotes the weight between the hidden layer and the output layer. The hidden layer unit is shown as z. The suffix i, j represents the units of the input layer and hidden layer respectively.

$$\Delta w_{1ij} = \eta(r - y)y(1 - y)w_{2j1}z_j(1 - z_j)x_i \tag{2.20}$$

$$\Delta w_{2j1} = \eta(r - y)y(1 - y)z_j \tag{2.21}$$

2.3.7 Optimization Algorithms

In the training of neural networks, optimization algorithms are used. Batch Learning, Sequential Learning, Mini-batch Learning, Stochastic Gradient Descent, and so on are some examples.

In each training epoch, batch learning traverses all of the training data, which takes much time in training data. The Equation 2.10 or another functions is used to determine the error of each training data E_n^t. The error of all the training data E^t in an epoch t is the total of E_n^t, as the Equation 2.22 shows. The adjusted weight of the network is fine-tuned based on all the training data for one of the network's training epochs. After then, this weight tests all of the training data. Every epoch, the network is corrected. The Equation 2.23 represents the total weight of the neural network, $w^{(t+1)}$.

$$E^t = \sum_{n=1}^{n} E_n^t \tag{2.22}$$

$$w^{(t+1)} = w^t - \eta \cdot \frac{\partial E^t}{\partial w^t} \tag{2.23}$$

The training data is entered one by one in Sequential Learning. It sees $E^t = E_n^t$ and fine-tunes the adjusted weight. The adjusted weight from the previous epoch is used to test the next training data and determine the new adjusted weight in the following epoch. Sequential Learning requires the training data does not have much difference or the epoch's result will not converge.

The training data is divided into sub-datasets through Mini-batch Learning, and each sub-dataset is used in each epoch. The Equation 2.24 shows the Mini-batch Learning, where D is a sub-dataset. After traversing all of the sub-datasets, the adjusted weight is fine-tuned from the first to the last. Mini-batch Learning combines the advantages of Batch and Sequential Learning, reducing training time and reducing epoch outcome variability.

$$E^t = \sum_{n \in D} E_n^t \tag{2.24}$$

The Stochastic Gradient Descent (SGD) (Robbins 2007) use a portion of the training data to avoid the epoch's outcome becoming a local optimal solution. SGD encompasses Sequential Learning as well as Mini-batch Learning. In most cases, data is chosen at random to create a sub-dataset from the training data.

2.4 Convolutional Neural Networks (CNN)

2.4.1 The brief history and structure of CNN

Neocognitron was proposed based on David Hunter Hubel's study on the visual cortex of cats (Fukushima and Miyake 1982). The Neocognitron is a neural network model with a hierarchical structure. The Neocognitron will add more cells to learn if no cells respond to the input. Backpropagation is used to introduce Convolutional Neural Networks (CNN) after being inspired by Fukushima's work (LeCun et al. 1989). Convolution layer, pooling layer, fully-connected layer, and output layer make up CNN. The Figure 2.9 shows the structure of CNN.

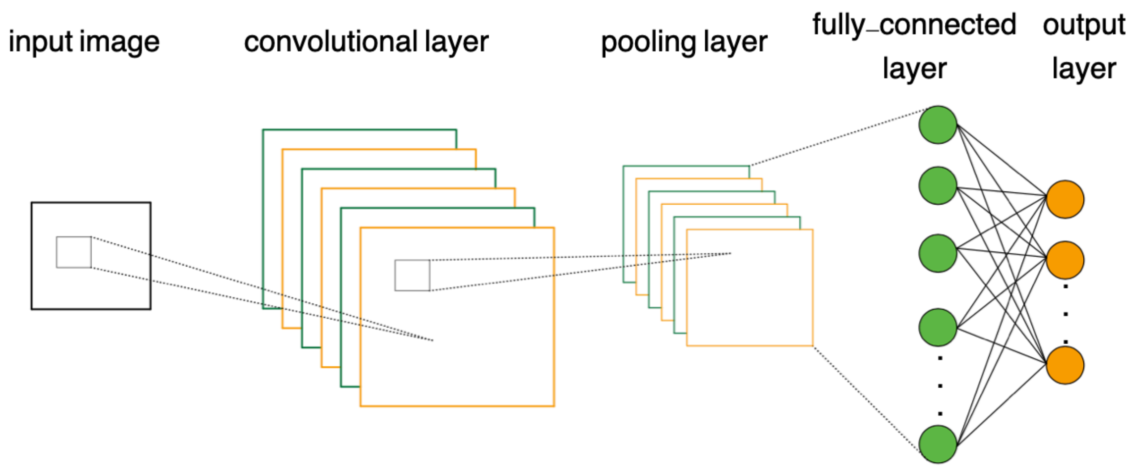


Figure 2.9 *the structure of CNN*

2.4.2 Convolutional layer

The input image is processed by the convolutional layer using the kernel to do convolution. When the input data is received by the first convolutional layer, the kernel performs convolution to obtain the feature map, which is then passed on to the next layer. The feature map created by the previous layer is processed by all subsequent convolutional layers. The following Figure 2.10 shows the convolutional layer with one channel kernel. In this figure, the orange part is the selected area which will does convolution with the kernel. The result is processed by the activation function and put the upper-left corner of the feature map.

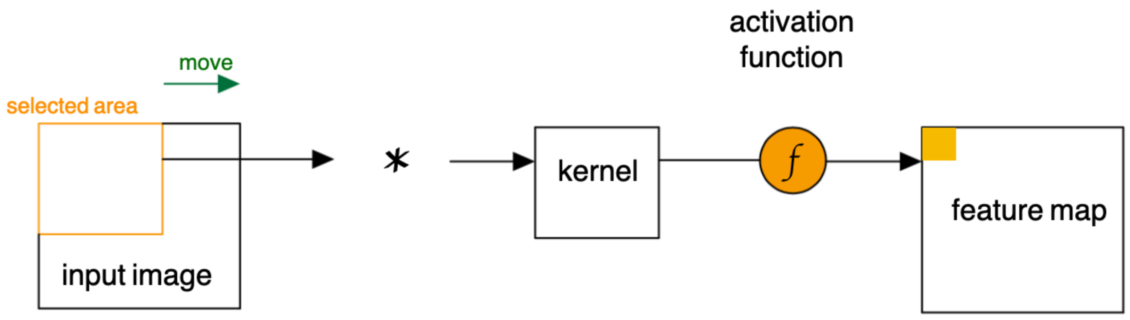


Figure 2.10 *the convolutional layer with one channel kernel*

The 3×3 kernel convolutions the 10×10 input image to produce the 8×8 feature map. Padding is used to process the input image first and then do the convolution, e.g. zero-padding adds 0 around the edges of the image to keep the size of the feature map the same as the size of the input. During the convolution process, each pixel value on the kernel multiplied another value on the input image at the corresponding place. The result of each multiplication is then added together to create a new value. The updated value is processed by an activation function, which will be displayed on the feature map.

Furthermore, stride has an impact on the size of the feature map. If the stride is greater than 1, the kernel will take a larger step on the input picture, reducing the size of the feature map. The result of the convolution is processed by the activation function and turn into a feature map.

There are several kernels in a convolutional layer. If the input image has three channels, the kernel will be $3 \times M \times M$, where $M \times M$ being the a kernel's size. n feature maps will be generated if there are n kernels. The kernel size of the next convolutional layer should be the same. As the following Figure 2.11 shows the convolutional layer with three channels kernel. The result of each channel is put on the upper-left corner of the feature map.

2.4.3 Pooling layer

The size of the feature map created by convolutional layers is reduced by using a pooling layer. The pooling window processes the input feature map in the same way as the convolution kernel, but instead of multiplying pixel values, it does other

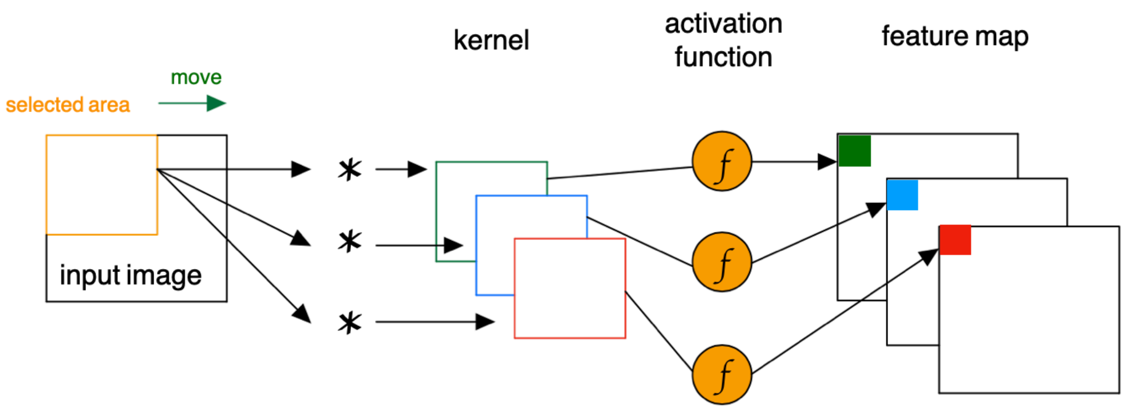


Figure 2.11 *the convolutional layer with three channels kernel*

pixel-by-pixel operations such as average, maximum, and so on. The Max Pooling layer finds the highest value in a given area and puts it to the new feature map. The Average Pooling Layer determines the average of the pixel values in the chosen area. The following Figure 2.11 shows the max and average pooling layer. In this figure, the pooling window selects 2×2 area and do max or average pooling.

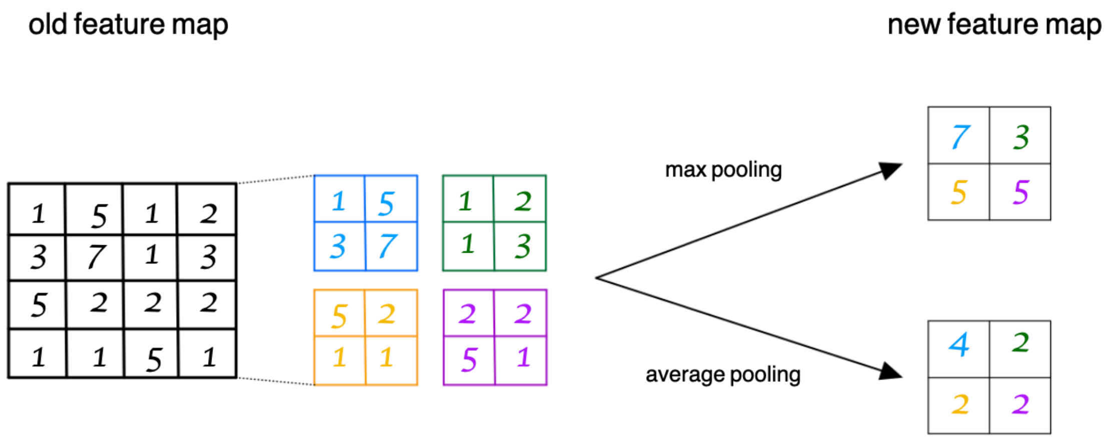


Figure 2.12 *the max and average pooling layer*

2.4.4 Fully-connected layer

The output from convolutional layers or pooling layers is received by the fully-connected layer. It first performs a linear transformation on the inputs before feeding them into the activation function to produce the final output. The green circles on the right part of the Figure 2.9 is the fully-connected layer.

2.4.5 Output layer

The likelihood of each category is calculated by the output layer of CNN using likelihood functions. CNN's final decision has the greatest chance of determining a category. *LeNet-5* was proposed in 1998 as a way to recognize handwritten numbers (Lecun et al. 1998). The number from 0 to 1 is referred to as the category in this model. If one of the categories has the highest likelihood, the model will output the number for that category. The orange circles on the right part of the Figure 2.9 is the output layer.

Softmax is used to calculate the likelihood in multi-classification issues. Its denominator adds together all of the output layer's units. As the following Equation 2.25 shows, the units are $q = 1, ..., Q$. Softmax selects the unit with the greatest probability as its output.

$$p(y^k) = \frac{exp(u_{2k})}{\sum_{q=1}^{Q} exp(u_{2q})}, \text{for } q = 1, ..., Q \tag{2.25}$$

2.5 Deep Learning Frameworks

Different deep learning frameworks models were applied in this thesis work. They were transformed to TensorFlow Lite so that Unreal Engine 4 (UE4) game engine can use them. The PyTorch model must first be converted to ONNX format before being imported into TensorFlow. The TensorFlow Lite converter can be used to convert the TensorFlow model.

2.5.1 TensorFlow

TensorFlow was derived from Google's DistBelief and was released on November 9, 2015. It's a end-to-end framework for implementing machine learning algorithms. It runs on Windows, Mac OS X, Linux, and other operating systems. TensorFlow includes a variety of deep learning APIs, such as vector and matrix calculations, optimization algorithms, CNN and RNN construction, and so on. With the help of TensorFlow, users can create, train, test, and save neural network models.

TensorFlow provides the following advantages over a variety of deep learning libraries: great flexibility and portability, multi-programming languages' support, and extensive documentation. First, TensorFlow, for starters, is extremely adaptable. TensorFlow graphs can be used to create networks. They can also use TensorFlow to construct their own libraries or APIs in Python or C++. Second, TensorFlow has high portability. It is run by a CPU or one or more GPUs. Personal PCs, servers, and mobile devices are used to run it. Third, TensorFlow supports some programming languages, e.g. Python, C++, R, etc. Users can run TensorFlow with the help of these languages' API. Forth, TensorFlow has extensive documentation published in a variety of natural languages, making it accessible to individuals all around the world.

TensorFlow's calculation is represented by a graph. There are "nodes" and "edges" in the graph. The operators are represented by "nodes," and the tensor is represented by "edges." The default graph in TensorFlow is $tf.get\_default\_graph()$, and users can design graphs using $tf.Graph$. To operate, the graph puts the calculation part to a $Session$. When TensorFlow is operating, the session controls all resources and releases them when the program is finished.

Tensor is the data type used by TensorFlow. It's a multidimensional array; for example, a zero-dimensional array is scalar, a one-dimensional array is vector, and so on. Name, shape, and type are the three characteristics of a tensor. The name of a tensor is a unique string; its shape is its dimension; and its type, such as *tf.int8*, is unique. Tensors are calculated using the same type of data.

On September 30, 2019, Google launched TensorFlow 2.0. TensorFlow 2.0 features the following main advantages over TensorFlow 1.0: easier to use, easier to build and deploy models, eager execution, and a simplified data pipeline. Some old libraries were deleted (e.g. $tf.contrib$) or combined; Eager Execution replaces $tf.Session$; Users can build models by Keras; TensorFlow Estimator API allows users run their models on local host or distributed multi-server environment; TensorFlow DataSet was introduced so users don't fed data to model by Placeholders.

2.5.2 PyTorch

PyTorch is a Facebook-developed open source deep learning framework. Torch, a library for calculating multidimensional matrices, is used to create it. PyTorch is a Python implementation of Torch. It has the properties of Lua-based Torch and GPU-based hardware acceleration.

PyTorch uses distributed training, a hybrid frontend, and libraries to make deep learning quicker and more customizable. It is fully integrated into Python and can be used in conjunction with Cython. Open Neural Network Exchange (ONNX) can also export the PyTorch model, allowing users to adapt it to other deep learning frameworks.

2.5.3 Open Neural Network Exchange (ONNX)

ONNX is an open format of deep neural network models. It is unrelated to system environment. Different AI models are interacted in a common format of ONNX. It supports many deep learning framework, e.g. PyTorch, TensorFlow, Caffe2, etc.

The ONNX format is comprised of three components. For starters, ONNX offers a definition for an extensible computation graph model, which dictates how graphs are represented in the intermediate stage. Second, ONNX offers definitions for common data types such as tensors, sequences, and maps. Third, ONNX contains built-in operator definitions. The default operators for neural networks, for example, are *ai.onnx*. Non-neural network machine learning models are represented by *ai.onnx.ml*.

2.5.4 TensorFlow Lite

TensorFlow Lite was launched by Google in May 2018. It's a deep learning framework for running TensorFlow Lite models on mobile, embedded, and Internet of Things devices. It has the following benefits: it is lightweight and cross-platform. TensorFlow Lite optimizes hardware acceleration and model load speed for mobile devices. It can swiftly initialize and run machine learning models on devices. It can also run on a variety of platforms, including Android and iOS.

The TensorFlow Lite interpreter and TensorFlow Lite converter are the two major components of TensorFlow Lite. A library serves as the interpreter. It has a simple API that allows mobile devices to run TensorFlow Lite models, as well as Linux devices. The Python API is used to convert TensorFlow models to TensorFlow Lite. It also makes binary files smaller.

As the following Figure 2.13, to use TensorFlow Lite, users should first convert a TensorFlow model to a TensorFlow Lite model in the form of *.tflite* using the TensorFlow Lite converter. This *.tflite* file might then be used on mobile devices. On Android, the Java API simplifies the encapsulation of C++ APIs. The TensorFlow Lite model is loaded and the interpreter is called using the C++ API. The model is run by the interpreter, which makes advantage of the Android neural network API to speed up the process.

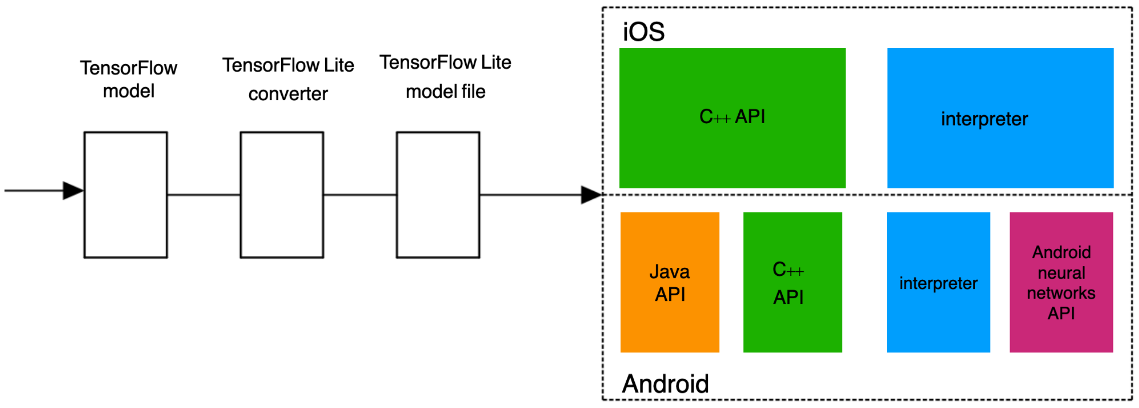

TensorFlow model
TensorFlow Lite converter
TensorFlow Lite model file
iOS
C++ API
interpreter
Java API
C++ API
interpreter
Android neural networks API
Android

Figure 2.13 *The architecture of TensorFlow Lite*

2.6 Unreal Engine

2.6.1 Unreal Engine 4 (UE4)

Epic Games, Inc, a video game and software firm based in the United States, created the Unreal Engine (UE4) game engine. It is utilized to create games and deploy neural network models in this thesis. This engine combines game development, simulation, and visualization features to fulfill a variety of game development needs.

UE4 game engine provides a lot of benefits. First and foremost, it excels at real-time rendering. Physically Based Rendering (PBR), Lighting Channels, Screen Space Reflection (SSR), and other abilities are greater in UE4. Second, UE4 introduces Blueprints Visual Scripting, a novel approach to script development. When compared to C++ code, it is easier for developers to define their own game logic events. The professional animation production is the third step. The Sequencer in UE4 is a set of non-linear and real-time animation tools. It contains a multi-track editor for creating cinematics in games. Fourth, UE4 includes robust game development frameworks, such as game rules, player I/O, camera and UI, and so on. It comes with a number of built-in games and character templates. Fifth, UE4 has advanced AI, such as Environment Query System (EQS), Behavior Trees, and so forth. Sixth, UE4 makes its source code available for developers to study.

2.6.2 Skeletal Mesh Animation

In UE4, Skeletal Mesh Animation System is a system for deforming skeletal meshes based on keyframed animation data and morph targets (Unreal Engine 4.27 documentation 2021). This section introduces the fundamental elements and process of this system.

Skeleton, skeletal mesh, and animation sequence are the three basic elements. The skeleton is the foundation of this system, and it contains information such as skeleton hierarchy, bone names, sockets, curves, and so on. The skeleton's mesh is referred to as skeletal mesh. It includes data such as Level of Details (LOD), Morph Target, Physics Asset, skin weight, and more. An animation component that records the movement of a skeletal mesh is called an animation sequence. In each frame of the animation, there are keyframes for the transformation of the bones. The skeletal mesh is animated by sequentially playing these keyframes.

To create skeleton mesh animation, you'll need the UE4 Animation Tools. Skeleton Editor is a tool used by developers to arrange bones into skeletons in a given hierarchy. The skeletons are then linked to their meshes using Skeletal Mesh Editor. After that, users create and alter animations in Animation Editor. The Animation Blueprint Editor creates the animation logic. The Physics Asset Editor is used to edit the physical bodies. In UE4, it calculates the pose of each bone, which is a collection of transforms (displacement, rotation, and scale). Following the calculation, UE4 does skinning, which involves deforming the vertices of a mesh based on the pose of the bone and the skinning weight.

2.6.3 Forward Kinematics and Inverse Kinematics

Forward kinematics (FK) are used in UE4 Animation Sequence. The parent node of a bone leads the child node to complete actions in this system. This method's calculation load is light, reducing the system's resource requirements. The child node of a bone, on the other hand, drives the parent node in the Inverse Kinematics (IK). It is appropriate for actions such as grabbing items and bending knees in response to the surroundings. The position of the target object is unpredictable in these scenarios. IK is preferable to FK for avoiding the need to create an animation sequence for each target object.

3 Human Pose Estimation Algorithms

Human pose estimation (HPE) can be divided into 2D HPE and 3D HPE. 2D HPE predicts horizontal coordinates and vertical coordinates of skeletal joints and in addition to x-axis and y-axis, 3D HPE predicts z-axis, the depth of the joints. In further, they can be divided into single pose estimation and multiple pose estimation. This thesis only built 3D human single pose estimation model because the application in this project is designed for a single human and 3D positions of joints are required for animating 3D character. 3D single pose estimation model can be classified into direct estimation approaches and 2D to 3D lifting approaches. Direct estimation approaches directly estimate 3D locations of joints from input 2D images and the latter leverage 2D predicted locations as intermediate representations to predict 3D location via 2D to 3D lifting model.

3.1 2D Pose Estimation

3.1.1 Neural network architectures

To learn dense prediction tasks, a High-Resolution Transformer (HRFormer) was presented (Yuan et al. 2021). To learn the high-resolution representation, this model was built based on HRNet. The HRNet, a high-resolution convolutional network, was built in a multi-resolution parallel approach (Wang et al. 2019). HRFormer improves the work of Vision Transformer (ViT) in ImageNet classification jobs (Dosovitskiy et al. 2020). HRFormer, in comparison to ViT, can model multi-scale variation and reduce feature granularity loss.

The following are some of HRFormer's features. HRFormer starts with the first stage of convolution and the three streams: high-resolution, medium-resolution, and low-resolution. Second, HRFormer maintains a high-resolution stream throughout the entire procedure. The representation of the high stream is improved by the other parallel streams. Third, HRFormer combines short-term and long-term attention.

As the Figure 3.1 shows, a representation map is separated into four non-overlapping portions in the HRFormer block. Multi-head self-attention (MHSA) is used to process each part, which improves memory and reduces computation complexity. However, there is not information exchange between these windows. The information is crucial since it aids in the expansion of the receptive field. As a result, the MHSA results are combined and fed into a feed-forward network (FNN) with 3×3 depth-wise convolution. This method improves the transformer's localisation.

The transformer modules and blocks in each module (except for the first stage module) all have a parallel construction, referenced the work of HRNet, as shown

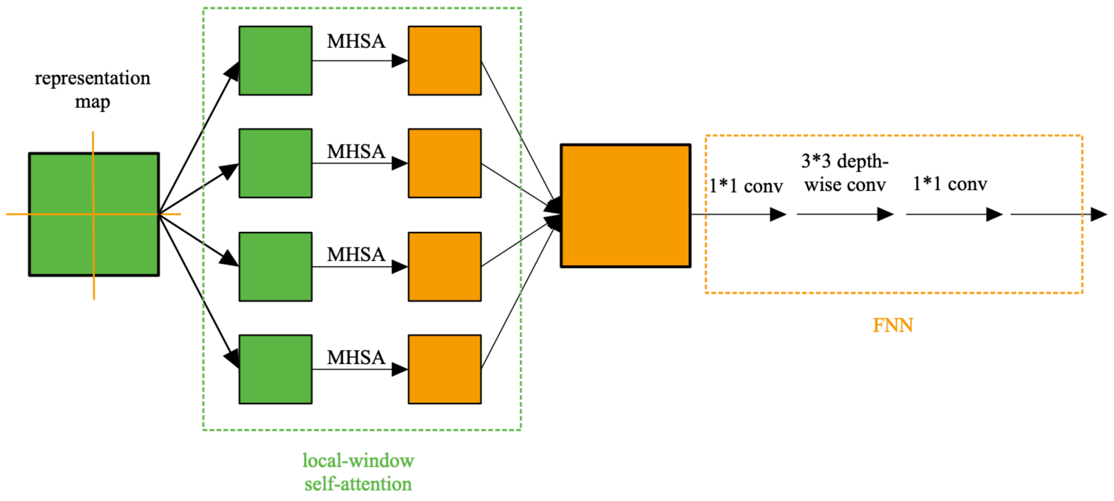


***Figure* 3.1** *the HRFormer block*

in Figure 3.2. As a transformer module, there is a high-resolution convolution block in the first stage. There are two blocks in the second stage: a high-resolution block and a medium-resolution block. The blocks are gradually added from the highest to the lowest in the following steps. Each stage updates the feature representation of each block. In a transformer module, information is exchanged among the blocks.

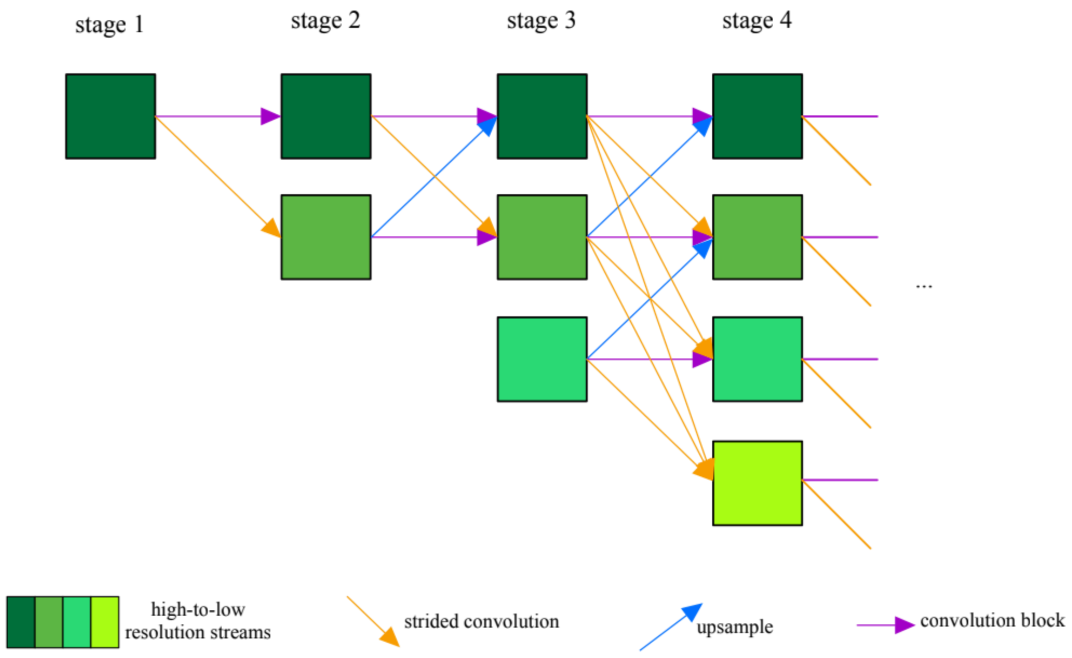


***Figure* 3.2** *the architecture of HRFormer*

To tackle the issues of human pose estimation remaining constrained by computational resources, Lite-HRNet was introduced (Yu et al. 2021). Their work increases the efficiency of high-resolution models.

The Lite-HRNet is made up of the high-resolution stem and the main body, as shown in Figure 3.2. The stem is the first stage. The stem has a 3×3 convolution with stride 2 and a shuffle block in the first stage. The main body is a sequence of modules and each module represents each stage (except the first stage). In each module of the main body, there are 2 conditional channel weighting blocks and 1 multi-resolution fusion. In the fusion, the streams are gradually added from the highest to the lowest. As the Figure 3.3 shows, in the conditional channel weighting block, the weight map is computed in each module across multiple resolutions. This calculation is called ”cross-resolution weight computation”. In each resolution, the spatial weights are computed.

This paper combines the shuffle block from ShuffleNet V2 (Ma et al. 2018), with the HRNet high-resolution stem to create a naive lightweight network. The shuffle block takes the place of the initial 3×3 convolution as well as the remainder of the residual block. The separable convolution substitutes the usual convolutions in the multi-resolution module of the main body (Chollet 2016). The naive lightweight network is replaced by the naive Lite-HRNet.

To reduce the complexity of channel weighting from quadralic to linear, the pointwise (1×1) convolution is replaced with a ligitweight unit called ”conditional channel weighting” in a shuffle block. The weights in this replacement are calculated using lightweight units throughout the multi-resolution channels. As a result of the weights, information can be exchanged between channels.

MobileNetV2 was proposed for mobile use in environments with limited resource (Sandler et al. 2018). It applies depth-wise separable convolutions, linear bottlenecks and inverted residuals. This model enhances the efficiency of mobile models, such as spectrum of models, benchmark, and so on, by reducing the number of operations and memory used while maintaining the accuracy of the model.

A factorized version of a full convolutional operator is replaced by two convolutional layers: a depth-wise separable convolution and a pointwise convolution. Each input channel is filtered in the former convolution. In order to create new features, the latter calculates the linear combination for those channels. The depth-wise separable convolution can reduce computation as compared to a typical convolutional layer.

The application of the inverted residual with linear bottleneck as a layer module is a highlight of this paper. The researchers decided to employ bottleneck convolution in MobileNetV2 because the linear layer has all of the information and can prevent non-linearity. This module gets the compressed representation in lower dimension as its input. Then, it expands the input to a higher dimension and makes it filtered by lightweight depth-wise convolution. The features are projected back

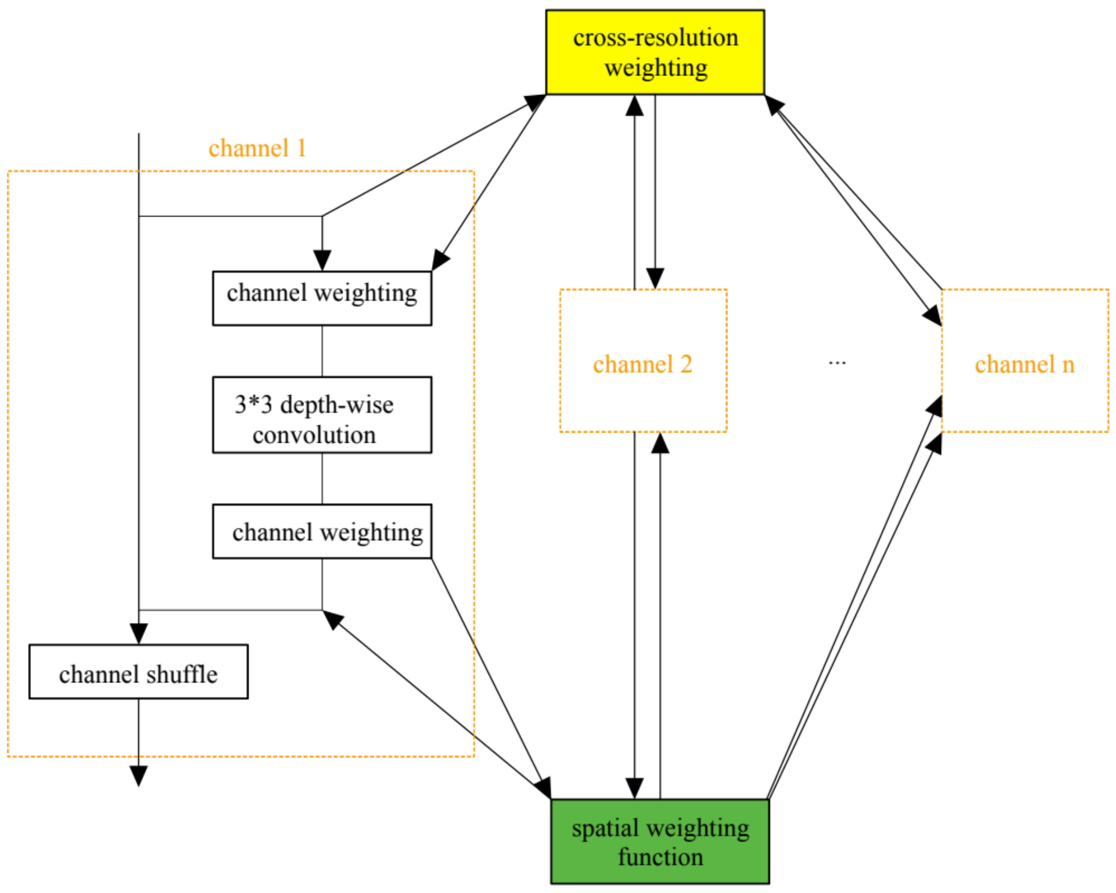


***Figure 3.3** the conditional channel weighting block of Lite-HRNet*

to the input. To promote gradient propagation across numerous layers, they use shortcuts between bottlenecks.

The bottleneck depth-separable convolution with residuals is employed as the main building block in the MobileNetV2 architecture. To keep the performance curve obvious, the constant expansion rate is between 5 and 10. As the Equation 3.1 reveals, the activation function is ReLU6 to improve the robustness of low-precision computation. The convolution block of MobileNetV2 is similar as the FNN part in the Figure 3.1. The input is processed by two convolution layer and one 3×3 depth-wise layer in the middle of them.

$$ReLU6(x) = \min(\max(0, x), 6) \tag{3.1}$$

The stacked hourglass (Hourglass) is made up of multiple stacked hourglass modules (Newell, Yang, and Deng 2016). Bottom-up and top-down inference is possible with these modules. The goal of this model is to determine the pixel location of the body's keypoints. The features are processed and consolidated in their model to acquire the body's spatial relationships. Across all image scales, this procession and consolidation are completed. Bottom-up, top-down, and intermediate supervision all help to improve the model's performance.

Hourglass is built on a foundation of pooling and upsampling in a sequential order. As an hourglass output, they generate a set of predictions. This paper creates a stacked hourglass after first implementing an hourglass.

Hourglass' goal is to make pixel-by-pixel forecasts of the body's keypoints. It reduces the features to 4×4 pixels in order to compare them with smaller spatial filters. The features will be upsampled to a greater resolution after that. These features will be blended at various resolutions. The hourglass is built in the following steps: initially, the convolutional and maxpooling layers reduce the resolution of the features to a very low level. Second, to merge features across scales, the top-down sequence is upsampled. Third, Tompson's work references the nearest upsampling of lesser resolution Tompson et al. 2014. Fourth, if the input resolution is equal to the output resolution, the model's predictions will be generated by a series of 1×1 convolutions. Hourglass generates heatmaps, one of which shows the probability of joint occurrence at each pixel.

An hourglass module has a symmetrical construction. The layers that are descending match to the layers that are ascending one by one. A residual module is represented by each block in this structure. The hourglass process is illustrated as the Figure 3.4 follows: the network begins with an 7×7 convolutional layer with stride 2. The output of the first step is then processed by the residual module in the second phase. Third, the max-pooling layer reduces the resolution to 64, which is quite low. There are two remaining modules at the fourth place. The output is generated in the fifth step. The output has 256 features and an 64×64 resolution.

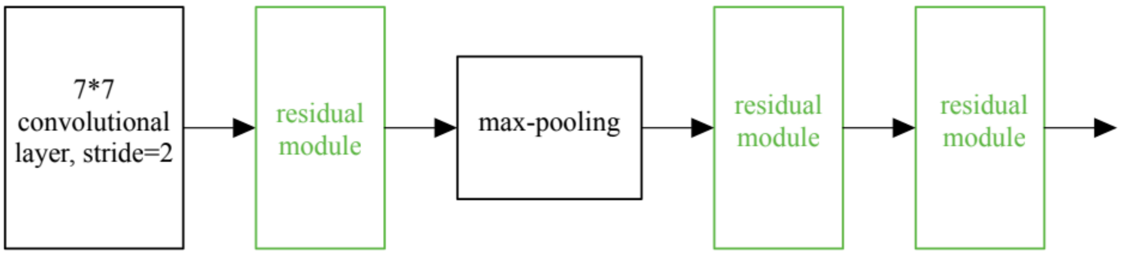


***Figure** 3.4 The part of hourglass*

The stacked hourglass (Hourglass) is completed by stacking many hourglasses, so the network will have the mechanism for repeated bottom-up and top-down inference.

3.2 3D Pose estimation by end-to-end network

The top-down approach was utilized to predict volumetric heatmaps for each skeletal joints by neural network and then got the 3D coordinates of these joints from heatmaps (Sangbum Choi, Seokeon Choi, and Kim 2021; Moon, Chang, and Lee 2019). Although they were estimated multi-person human pose, they are applied in single-person pose estimation and their performance exceed many other models for single-person pose estimation. The following will introduce the neural network architectures and the algorithm of generating 3D coordinates from volumetric heatmaps.

3.2.1 Overall pipeline

As Figure 3.5 shown, this pipeline consists of three neural networks, DetectNet, RootNet and PoseNet. DetectNet is responsible for detecting humans and generating bounding box for each human body from the input image and cropping the input image based on the bounding box to make each sub-image only contain single person, then applying RootNet and PoseNet to cropped images respectively. RootNet is used for predicting camera-centered coordinates of the detected human's root, a reference point in body, such as pelvis. PoseNet is used for estimate root-relative 3D single-person pose and then output the 3D multi-person pose by adding root coordinates and each root-relative skeletal joints coordinates.

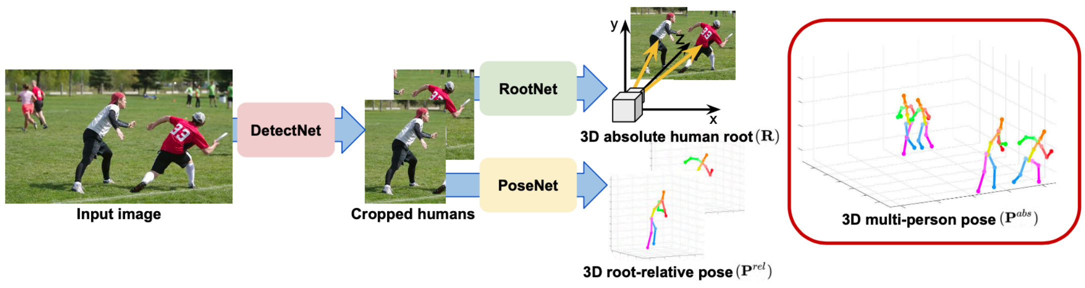

Input image
DetectNet
Cropped humans
RootNet
PoseNet
y
x
3D absolute human root ($\mathbf{R}$)
3D root-relative pose ($\mathbf{P}^{rel}$)
3D multi-person pose ($\mathbf{P}^{abs}$)

Figure 3.5 *Overall pipeline propose for multi-person 3D pose estimation (Sangbum Choi, Seokeon Choi, and Kim 2021)*

Other approach often used in pose estimation is called bottom-up approach, which is predicting all the body keypoints, no matter which person these keypoints belong to and then group them into each person using clustering techniques.

3.2.2 Neural network architectures

In DetectNet and PoseNet, the same neural network frames were used while the new architectures for PoseNet to make network more efficient and have less computing cost, which is suitable for resource limited device, such as phones and suitable for the application that requires less time to predict results, such as games as well (Sangbum Choi, Seokeon Choi, and Kim 2021; Moon, Chang, and Lee 2019).

The framework of DetectNet used (Sangbum Choi, Seokeon Choi, and Kim 2021 and Moon, Chang, and Lee 2019), is Mask R-CNN (He, Gkioxari, et al. 2018). Mask R-CNN includes a backbone to extract features from the input image via deep residual network (He, Zhang, et al. 2015) and pyramid network (Lin, Dollár, et al. 2017), a region proposal network to propose body bounding box candidates and then extract the features of each proposal bounding box, and finally, a head network to output the refined bounding box of human body and probability if the input image contains the human body. This architecture was used as their DetectNet (Sangbum Choi, Seokeon Choi, and Kim 2021; Moon, Chang, and Lee 2019) because high performance was achieved (He, Gkioxari, et al. 2018) on several public object detection datasets (Lin, Maire, S. Belongie, et al. 2015).

Sangbum Choi, Seokeon Choi, and Kim 2021 proposed a new network architecture as RootNet to estimate 3D coordinates of human root $R = (x_R, y_R, z_R)$, as Figure 3.6 shows that it predicts depth value z_R (the distance between camera and human root) and 2D coordinate (x_R, y_R) respectively. First, a backbone network (ResNet) extracts image features from the input image and the features are passed to two networks respectively, 2D pose estimation network and depth estimation network (He, Zhang, et al. 2015). 2D pose estimation network includes a upsampling layer, three consecutive deconvolutional layers with batch normalization layers (Ioffe and Szegedy 2015) and ReLU activation layer, and a 1X1 convolution layer to produce a 2D heatmap of the human root. Then soft-argmax is applied to generate 2D coordinates x_R, y_R from a 2D heatmap (Sun et al. 2018).

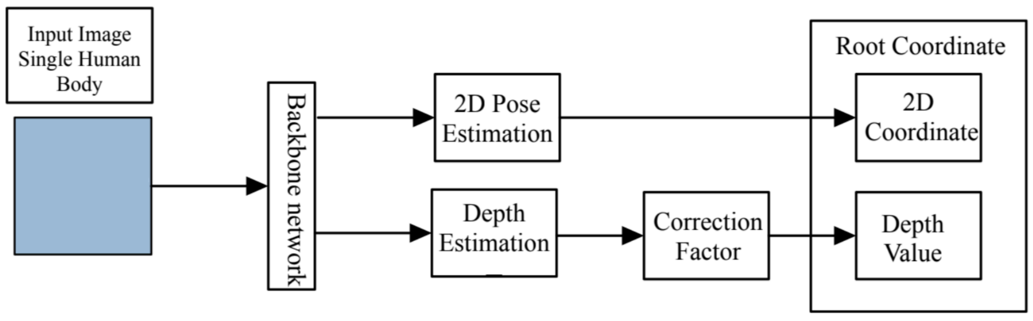


Figure 3.6 *RootNet framework proposed (Moon, Chang, and Lee 2019)*

The Depth Estimation part estimates the distance from camera to the root, which ie calculated by Equation 3.2, where α_x, α_y are the focal lengths divided by the per-pixel distance factors of *x-axes* and *y-axes*. I_{real} is the area of human body in the real space (mm^2) and I_{image} is the area of human body in the image space ($pixel^2$).

$$d = \sqrt{\alpha_x \alpha_y \frac{I_{real}}{I_{image}}}, \tag{3.2}$$

Equation 3.2 is be derived from a pin-hole camera model, which can be visualized in Figure 3.7, where d is the distance between camera and human root (mm), f is the focal length of the camera (mm), H_{real}, W_{real}, H_{sensor}, W_{sensor} are the heights and widths of the human body in real space and on image sensor respectively.

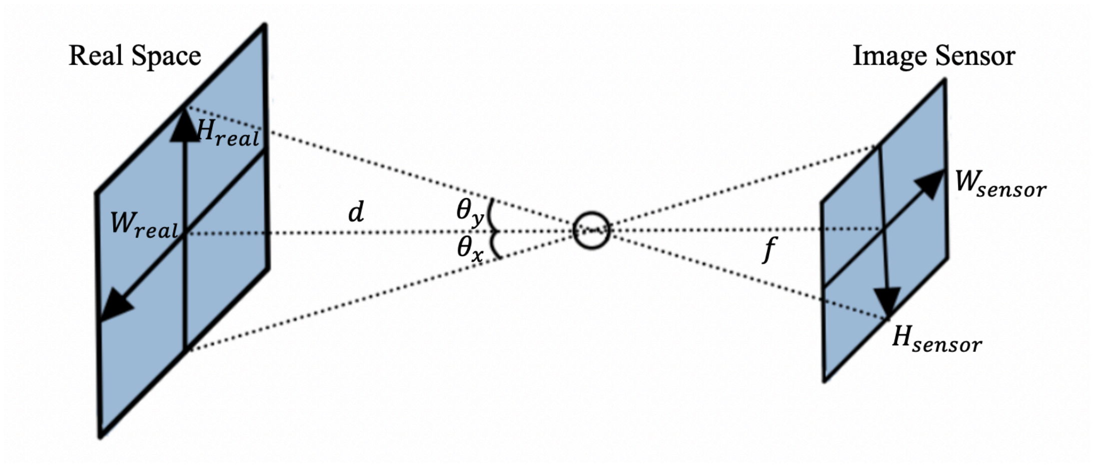


Figure 3.7 *Visualization of pinhole camera model*

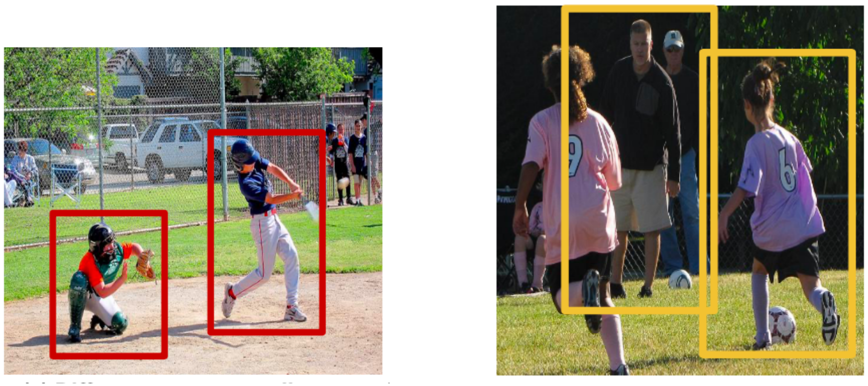
According to the definition of tan,

$$tan\theta_x = \frac{W_{real}}{2d} = \frac{W_{sensor}}{2f}, \tag{3.3}$$

$$tan\theta_y = \frac{H_{real}}{2d} = \frac{H_{sensor}}{2f}, \tag{3.4}$$

As the Equation 3.5 shows the distance d, where p_x, p_y are the per-pixel distance factors for *x-axes* and *y-axes*.

$$\begin{aligned} d &= f\frac{W_{real}}{W_{sensor}} = f\frac{H_{real}}{H_{sensor}} \\ &= fp_x\frac{W_{real}}{p_x W_{sensor}} = fp_y\frac{H_{real}}{p_y H_{sensor}} \\ &= \alpha_x\frac{W_{real}}{W_{image}} = \alpha_y\frac{H_{real}}{H_{image}} \end{aligned} \tag{3.5}$$



(a) Different measured distances (b) Same measured distances

Figure 3.8 *Examples of wrong measured distances by Equation 3.2 (Moon, Chang, and Lee 2019)*

Then we can get Equation 3.2 by taking square root of Equation 3.6.

$$
\begin{aligned}
d^2 &= d_x d_y \frac{W_{real} H_{real}}{W_{image} H_{image}} \\
&= \alpha_x \alpha_y \frac{I_{real}}{I_{image}}
\end{aligned}
\tag{3.6}
$$

However, directly using d as the depth value z_R leads to mistakes in several cases. For example, as Figure 3.8(a) shows that the measured distances from Equation 3.2 of two people are different although the true distances should be same due to the size of bounding box while Figure 3.8(b) shows that measured distances of two people are same, but the true distances should be different. To correlate it, a depth network was trained to generate correlation factor γ (input is the image feature extracted from a backbone network) (Sangbum Choi, Seokeon Choi, and Kim 2021), then $k/\sqrt{\gamma}$ is regarded as the final depth value Z_R.

The PoseNet in this pipeline derives from the model (Sun et al. 2018) and then we can get the volumetric heatmaps from the PoseNet. Based on the pipeline (Moon, Chang, and Lee 2019; Sangbum Choi, Seokeon Choi, and Kim 2021), a new network for PoseNet was proposed, which has better trade-off between quality and time. The network architecture consists of a convolution layer with batch normalization and PReLU activation layer, seven inverted residual blocks and three deconvolution blocks and a convolution layer as final layer. Specifically, two different types of concatenation structures were proposed, residual concatenation and skip concatenation (Sangbum Choi, Seokeon Choi, and Kim 2021). The both types can be visualized in Figure 3.9 and Figure 3.10 respectively. Comparing with the residual concatenation structure, skip concatenation structure increases the estimated quality without decrements of efficiency.

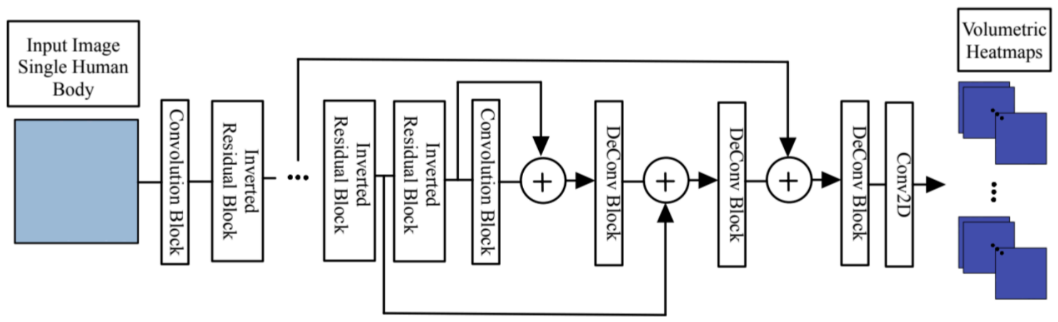


Figure 3.9 *Skip concatenation structure used in MobileHumanPose (Sangbum Choi, Seokeon Choi, and Kim 2021)*

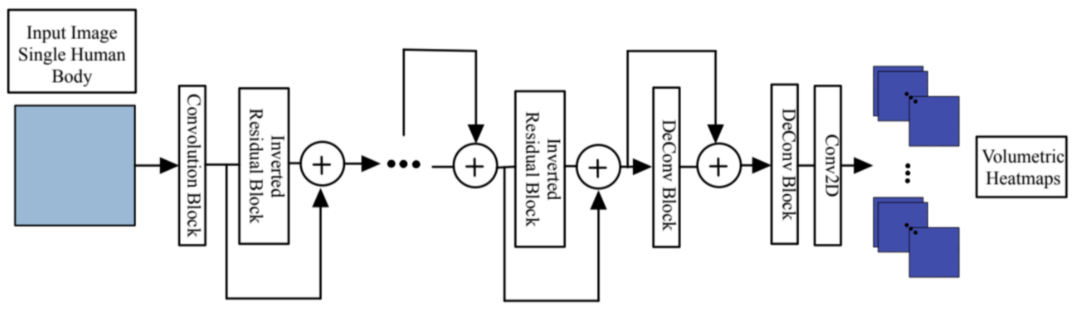
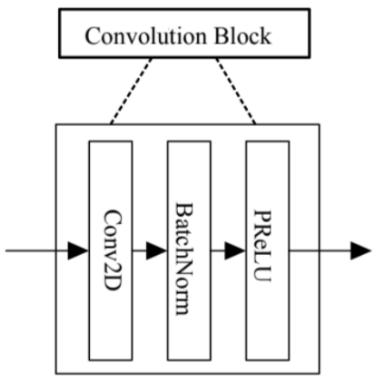


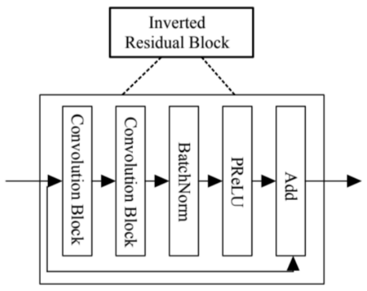
Figure 3.10 *Residual concatenation structure used in MobileHumanPose (Sangbum Choi, Seokeon Choi, and Kim 2021)*

The Convolution block is a convolution layer followed by a batch normalization and PReLU activation function, where PReLU is parametric ReLU (see Equation 3.7) and it has been shown that the performance increases when PReLU is used as activation function in 2D human pose estimation (Bulat et al. 2019). The inverted residual block contains two consecutive convolution blocks followed by a batch normalization and PReLU activation function. The deconvolution blocks contains a two convolution layers with batch normalization and PReLU function and a upsampling layer. These three structures are shown in Figure 3.11, where a_i is a learnable parameter and y_i is a input signal.

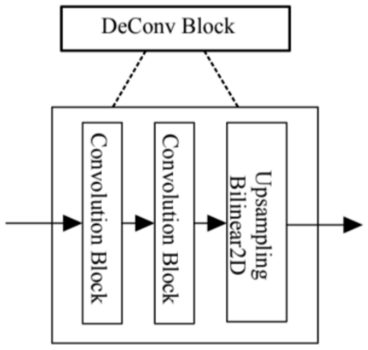
$$f(y_i) = \begin{cases} y_i, & \text{if } y_i > 0 \\ a_i y_i, & \text{if } y_i \leq 0 \end{cases} \tag{3.7}$$



(a) Convolution block



(b) Inverted residual block



(c) Deconvolution block

Figure 3.11 *Structures of convolution block,inverted residual block and deconvolution block used in MobileHumanPose (Sangbum Choi, Seokeon Choi, and Kim 2021)*

3.3 3D Pose Estimation by Lifting Approach

Lifting approach consists of two stages: image-to-2D-pose Network and 2D-to-3D Network. Compared to end-to-end networks, lifting method is much simpler and faster and more easily to understand and reproduce (Martinez et al. 2017).

To estimate the locations of human body joints in 3-dimension space, $\mathbf{x_i}$, they employed an array of points in 2-dimension space, $\mathbf{y_i}$. To maintain the prediction accurate, As the Equation 3.8 shows, they should keep the prediction error to a minimum, where, $\mathbf{x_i} \in \mathbb{R}^{2n}$, $\mathbf{y_i} \in \mathbb{R}^{3n}$; L is loss function; f^* denotes a deep neural network.

$$f^* = \min_f \frac{1}{N} \sum_{i=1}^{N} L(f(\mathbf{x_i}) - \mathbf{y_i}) \tag{3.8}$$

As the following Figure 3.12 shows, the architecture of their model is a multi-layer neural network with a simple and deep design. They picked $2-d$ points as input for the following reasons: training time will be reduced; network training and design will be accelerated; and the dataset is readily stored on GPU during training.

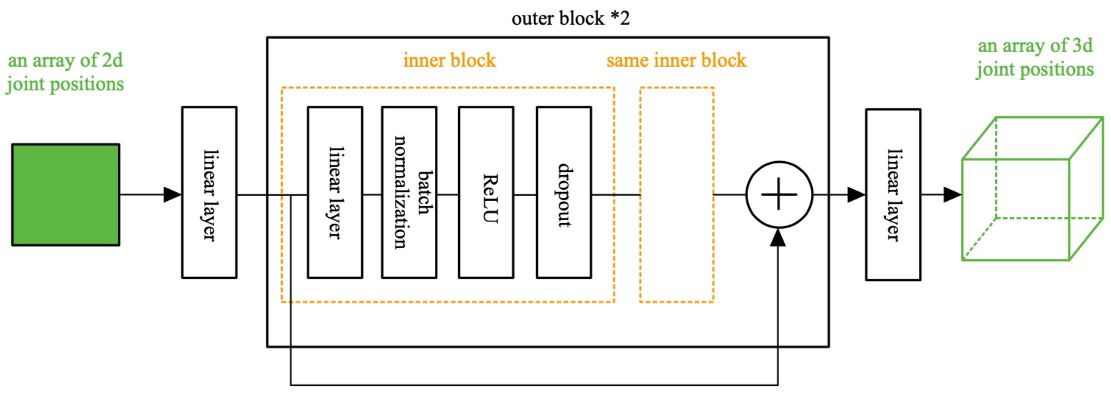


Figure 3.12 *The architecture of the lifting approach model*

There are 6 linear layers in the architecture.The first and last linear layers are placed after and before the input and output, respectively. The first layer increases the input dimension to 1024, while the last layer reduces the output size to $3n$. The remaining 4 linear layers are added to the building blocks which repeated for two times.

Two structures are repeated in a building block. A linear layer, batch normalization, ReLU, and dropout are all present in each structure. To solve this model's poor performance in training the $2-d$ detector's output, training $2-d$ ground truth, and assessing noisy $2-d$ observations, batch normalization and dropout are added. Between batch and dropout, the ReLU is applied, because the input and output are low-dimensional, it was decided to include nonlinearity in their model. Weights are added constraints for each layer in the block to keep the training stable and increase generalization in situations when training and testing data have distinct distributions.

4 Experiments

This thesis trained and tested different pose estimation models in terms of prediction accuracy, computation complexity and inference time in a real-time application. For the training part, COCO (Microsoft Common Objects in Context) dataset is used for training and testing 2D pose models because these datasets contains images and 2D coordinates of landmarks, and training and testing 3D pose models is on Human3.6M dataset, containing 2D coordinates of landmarks or images as input and 3D coordinates as ground-truth, which can be utilized to train 2D-to-3D lifting model and 3D pose models.

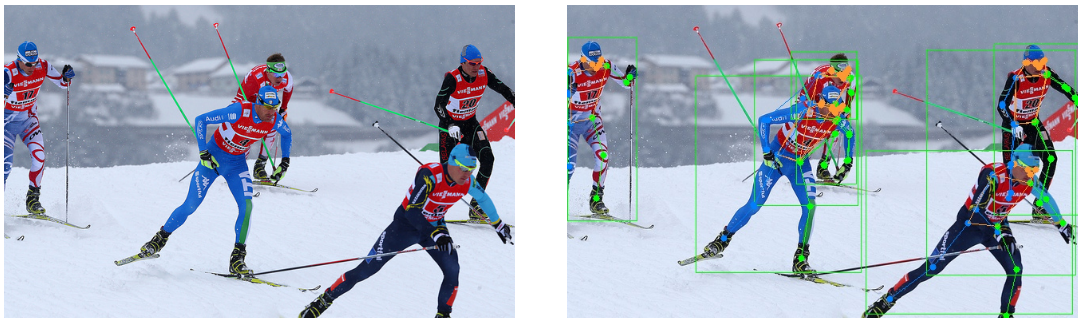
4.1 Dataset

4.1.1 Microsoft Common Objects in Context

The MS COCO (Microsoft Common Objects in Context) dataset was created (Lin, Maire, S. J. Belongie, et al. 2014). It is used to aid in the detection and segmentation of items in their natural environment by extending object recognition to scene comprehension. There are 91 object categories in the MS COCO. 82 of these categories have more than 5000 labeled instances. In this dataset, there are 2.5 million labeled instances in 328000 images. These labeled cases help in the model's acquisition of contextual knowledge.

By distinguishing between "staff" and "object," this dataset selects common categories. The term "staff" refers to objects that have no obvious borders. The easily named things are the "thing." The "staff" is dropped in favor of the "thing" in order to get precise object instance localisation. In addition, MS COCO's categories are all entry-level. Based on the types of images, this dataset selects non-iconic images. In the case of iconic-object and iconic-scene photographs, the quality is excellent, but there is less contextual information. Non-iconic images, the third type, are good at generalizing.

The MS COCO addresses three issues: first, detecting non-iconic views of objects; second, detecting non-iconic views of items; and third, detecting non-iconic views of objects. Existing object recognition systems are hard to distinguish objects in the background that are partially occluded by clutter (Hoiem, Chodpathumwan, and Dai 2012). The second issue is object-to-object contextual reasoning. Because the identities of items can be determined from their surroundings, images of scenes are more important than images of objects. The images in this dataset not only have contextual relationships but also have non-iconic object views. The exact 2D localisation of objects is the third issue. As the following Figure 4.1 show, the spatial location of a object is determined by bounding boxes. Every instance of each object's category is labeled and segmented in this dataset.



(a) the original image

(b) the image with its bounding boxes and keypoints

Figure 4.1 *The images in MS COCO (Lin, Maire, S. J. Belongie, et al. 2014)*

The researchers used the following steps to annotate images: The first step is to label the categories. The object categories were amalgamated into super-categories in order to speed up the classification process. Image spotting is the next phase. Crossings are used to designate all of the instances in an image. Instance segmentation is the third phase. The instances are divided into groups. Then they are subjected to verification to confirm that the segmentation was of high quality. If there are less than or equal to 10 instances of a category in the photos, the instances will have been segmented separately. The instances will be grouped if there are more than 5 instances. Finally, descriptive captions are written for each image.

4.1.2 Human3.6M

Using marker-based motion capture equipment, Human3.6M dataset was created and asked professional actors to perform a wide range of gestures in indoor areas (Ionescu et al. 2014). This dataset provides a wide range of human poses, motions, and activity scenarios. This dataset can be used to train human sensing systems and evaluate novel human posture estimation methods.

Human3.6M covers four aspects: For starters, this dataset is quite enormous. It includes over 3.6 million various human positions photographed from 4 perspectives. The everyday scenarios are accomplished by 11 professional actors. Second, the dataset contains data that is synchronized. Depth sensor, video cameras, and progressive scan synchronize 2D and 3D data in Human3.6M. Third, Human3.6M supports mixed-reality environments. Motion capture data is generated by inserting the characters into the video environment. Fourth, this dataset is utilized to estimate poses.

During the image capture, 15 sensors, like as motion cameras, video cameras, and time-of-flight sensors, collect the data in this dataset. Hardware and software keep them in sync. The actions are carried out by 6 actors and 5 actresses, and they belong to the Body Mass Index range of $[17, 29]$. 7 and 4 people complete the motions of the training and testing sets, respectively. As a result, the dataset includes a wide range of body types and mobility. Moreover, the data is in the parametrization of ”relative 3D joint positions” and ”kinematic representation”. Human3.6M includes a complete skeleton with 32 joints as well as bounding box annotations for humans in photos. Bounding boxes are derived from the projection of skeletons in images, which employ a bounding box to fit around the projection.

4.2 Training 2D and 3D pose estimation models

This thesis trained 5 different pose models, shown in the Table 4.1. The stacked-Hourglass, HRFormer, Lite-HRNet and MobileNetV2 are trained on COCO dataset, where the input is single image and bounding box of a human in the images and output is 2D coordinates (x and y) of 17 keypoints, consisting of nose, left eye, right eye, left ear, right ear, left shoulder, right shoulder, left elbow, right elbow, left wrist, right wrist, left hip, right hip, left knee, right knee, left ankle, right ankle, as the same order of the output from model, as Figure 4.2 shows. Additionally, these 2D models were trained 200 epochs with learning rate 0.0005 and the the training curves of MSE (mean squared error) loss of these 4 2D models can be seen in the Figure 4.3.

Table 4.1 *The trained models*

model	**input**	**output**	**datasets**
StackedHourglass (Newell, Yang, and Deng 2016)	single image	17 2D points	COCO
HRFormer (Yuan et al. 2021)	single image	17 2D points	COCO
Lite-HRNet (Yu et al. 2021)	single image	17 2D points	COCO
MobileNetV2 (Sandler et al. 2018)	single image	17 2D points	COCO
2D-3D Lifting (Martinez et al. 2017)	17 2D points	17 3D points	Human3.6M

The 2D-to-3D lifting model was trained on the Human3.6 dataset, containing 17 keypoints of human body, but the locations of these 17 keypoints (pelvis, right hip, right knee, right ankle, left hip, left knee, left ankle, torso, neck, nose, head, left shoulder, left elbow, left wrist, right shoulder, right elbow, right wrist), which is shown in the Figure 4.4. This model was trained 200 epochs as well with learning rate 0.001 and the Figure 4.5 shows the loss curve. According to the loss curve, where the loss is reduced rapidly at the first epoch and the curve is quite flat after couple of epochs, which means that the regression of 2D points to 3D points is

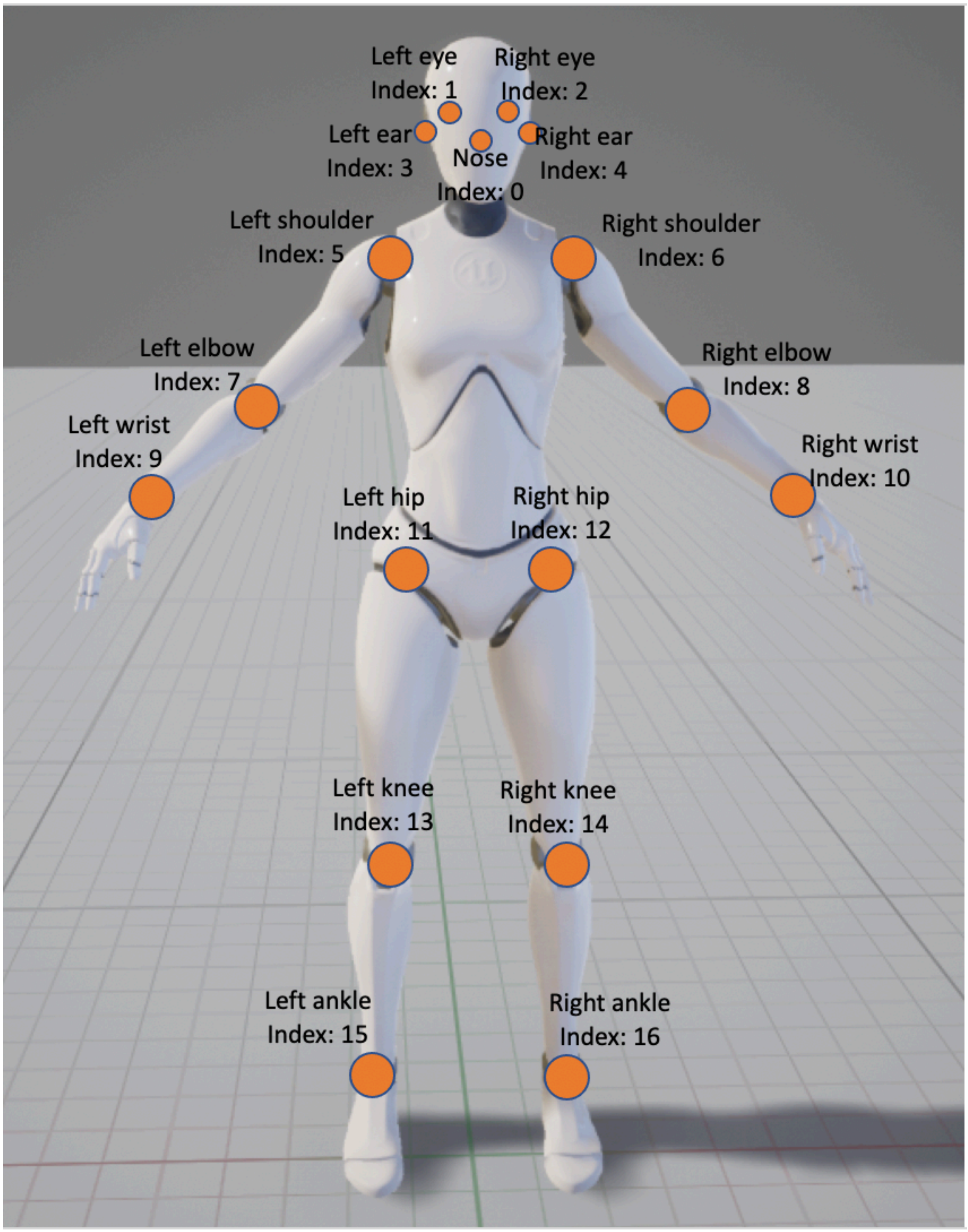
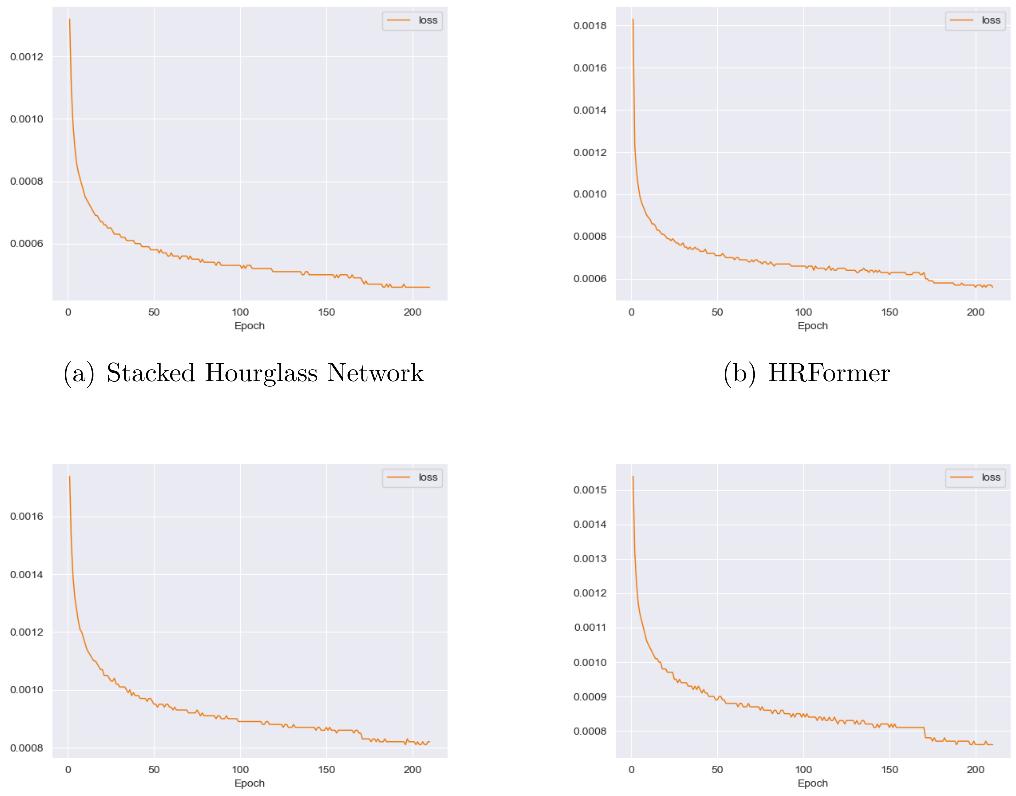


Figure *4.2 Keypoints and index of human pose in COCO dataset*



(a) Stacked Hourglass Network (b) HRFormer

(c) Lite-HRNet (d) MobileNetV2

Figure 4.3 *The curves of training loss of different models*

converged quickly and 2D-to-3D regression model can be trained only one epoch or several epochs for fast training. The Figure 4.6 shows the loss curve at the first epoch.

4.3 Test prediction accuracy of human pose of pose models

This section shows the prediction accuracy of 3D keypoints and some visualizations of predicted results on Human3.6 dataset from the models trained in the section4.2 and the pretrained MobileHumanPose model (Sangbum Choi, Seokeon Choi, and Kim 2021).

4.3.1 Inference pipeline

There are two stages for inference with the model, MobileHumanPose: detecting pelvis location using RootNet and predicting 3D points using PoseNet.

But generally, generating 17 keypoints from a single image required three stages: human detection, 2D points prediction and 3D points prediction. In the human

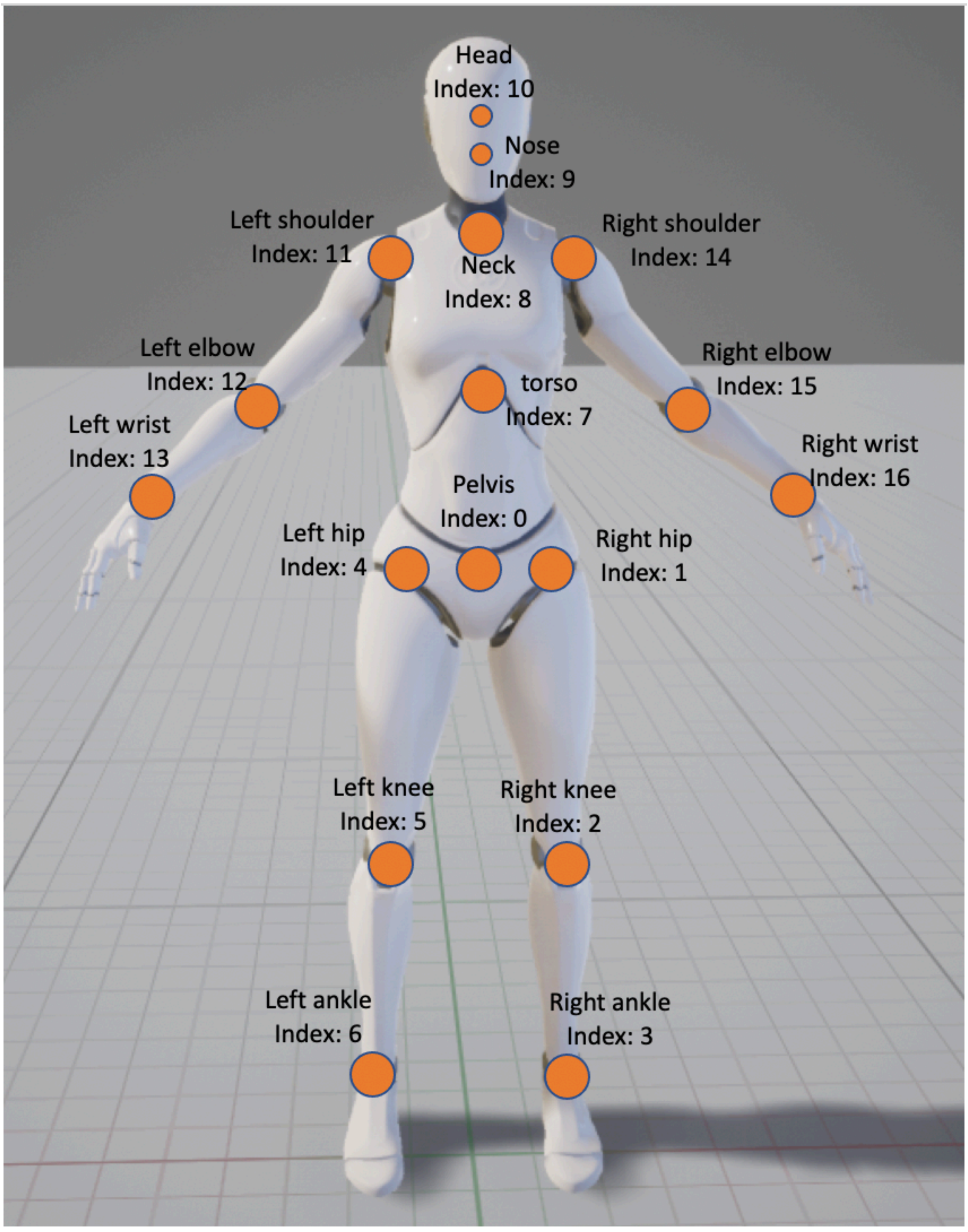


Figure *4.4 Keypoints and index of human pose in Human3.6 dataset*

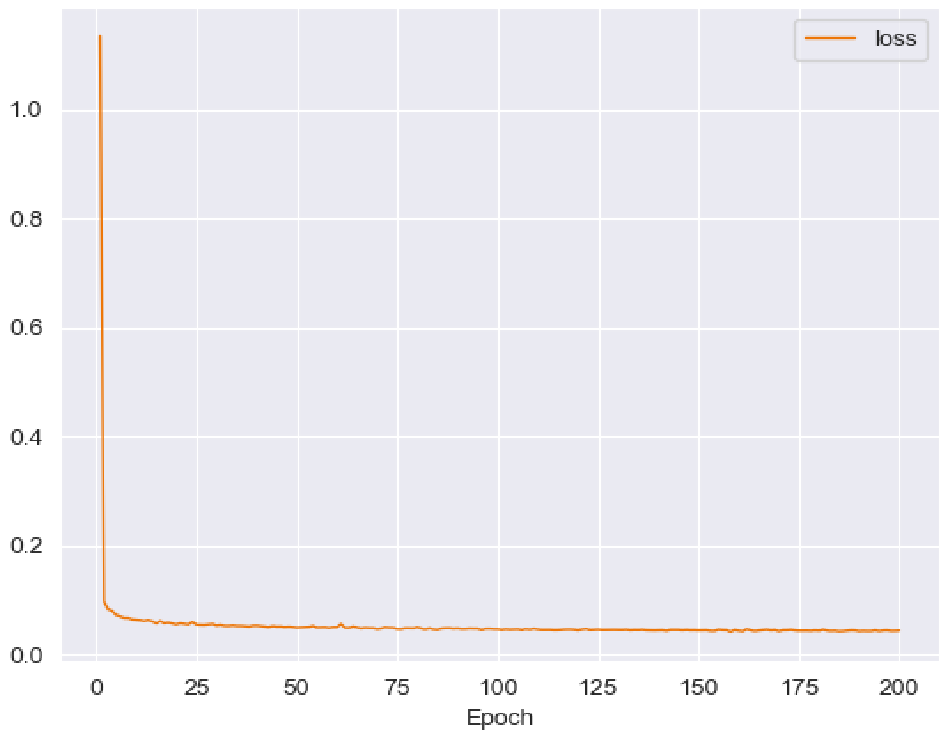


Figure 4.5 *The training loss curve of 2D-3D pose lifting model for 200 epochs*

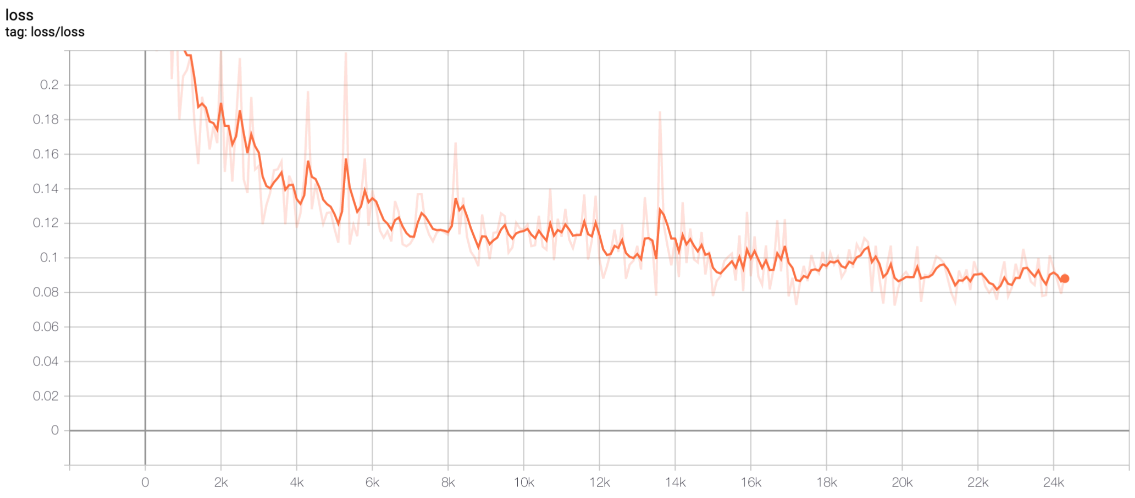


Figure 4.6 *The training loss curve of 2D-3D pose lifting model for 1 epoch*

detection stage, this thesis used the model YOLOv3 (Redmon and Farhadi 2018) to detect the location of human, represented as a bounding box. Then the cropped image within the bounding box is a input of 2D pose models and the 2D pose models was used to generate 2D points. Finally these predicted 2D points were passed to the 2D-to-3D lifting model to get 3D keypoints. But between the second and third stage, there was required a keypoint transforming process, because the indices of keypoints of datasets are different between COCO dataset and Human3.6 dataset, where 2D models were trained on the COCO dataset while lifting model is trained on Human3.6 dataset. The transformation method is described below:

First, pelvis, neck, head, torsor are in the Human3.6 but not in the COCO, but pelvis is in the middle of left hip and right hip; neck is in the middle of left shoulder and right shoulder; head is in the left eye and right eye; torsor is in the middle of neck and pelvis, then these points can be calculated as Equation 4.1 shows.

$$pelvis = \frac{left\,hip + right\,hip}{2}, \tag{4.1}$$

$$neck = \frac{left\,shoulder + right\,shoulder}{2}, \tag{4.2}$$

$$head = \frac{left\,eye + right\,eye}{2}, \tag{4.3}$$

$$torsor = \frac{neck + pelvis}{2}, \tag{4.4}$$

$$\tag{4.5}$$

Second, changing the index order. For example the index of left shoulder is 5 in the COCO dataset, but it's 11 in the human3.6 dataset, thus the order of index should be changed before passing 2D keypoints data into lifting model.

4.3.2 Prediction accuracy

The 2D pose models were evaluated based on distance accuracy, which represents the percentage of positive predictions. First, the distance between normalized predicted keypoints and normalized ground-truth was calculated, and if the value of distance was less than a threshold. i.e 0.5, we thought this prediction is positive and accuracy equals number of positive predictions divided by total number of keypoints. 3D pose models were evaluated based on MPJPE (Mean Per Joint Position Error), which is the Euclidean distance between coordinates of ground-truth and estimated keypoints as Equation 4.6 shows, where, $\mathbf{J_i}$ is the coordinates of ground truth of the $\mathbf{i}$th keypoint, $\mathbf{J_i^*}$ is the coordinates of $\mathbf{i}$th estimated keypoints and N denotes the number of keypoints.

$$MPJPE = \frac{||J_i - J_i^*||_2}{N} \tag{4.6}$$

The Figure 4.7 shows the accuracy of 4 different 2D pose, which is only evaluated on COCO datasets for 2D keypoints prediction. From the figure, stackedHourglass model shows the highest accuracy for 2D pose predicting. Furthermore, the 3D pose models (lifting model, stackedHourglass + lifter, HRFormer + lifter, Lite-HRNet + lifter, MobileNetV2 + lifter and MobileHumanPose) were tested on the Human3.6 dataset, and the MPJPE score is shown in the Table 4.2. From the table, the Lifting model has the lowest MPJPE score, which means Lifting model has the least prediction errors. This is because the input of lifting model is the ground-truth 2D keypoints while for other models, the input is the RGB images. The combination of stackedHourglass model and lifting model shows the best prediction results and the performances of most combination methods (2D model + lifting model) are better than the MobileHumanPose, which is the end-to-end 3D pose model.

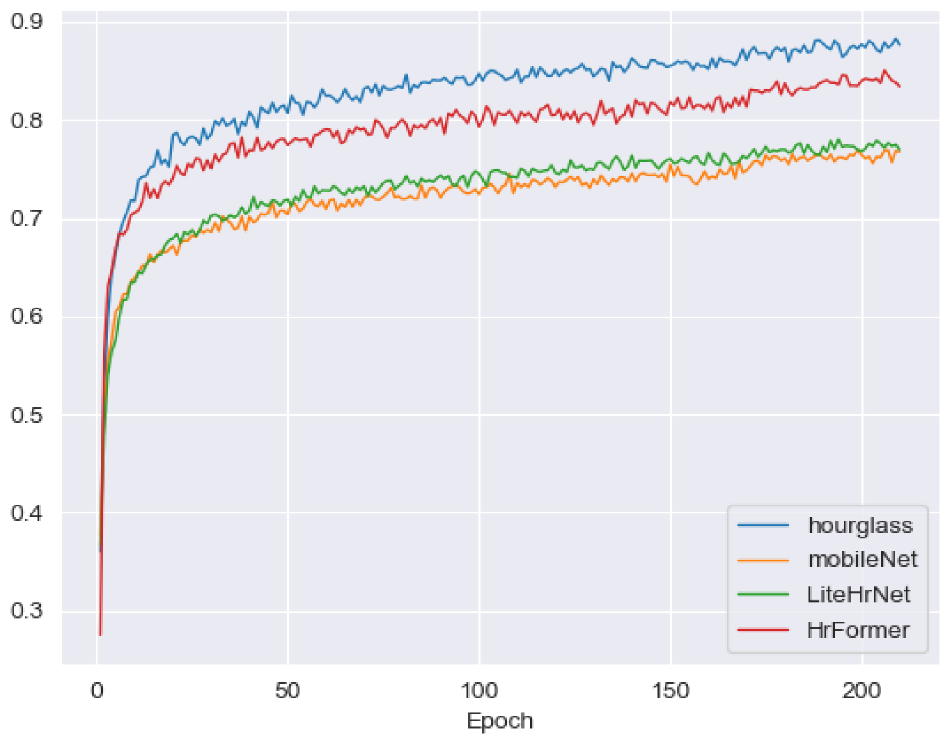


Figure 4.7 *Accuracy of pose prediction of different 2D pose models*

Table 4.2 *MPJPE score of different models tested on Human3.6 dataset*

model	**MPJPE(mm) score**
Lifting model(use 2D ground truth as input)	52.17
StackedHourglass+lifting	54.58
HRFormer+lifting	58.70
Lite-HRNet+lifting	58.72
MobileNetV2+lifting	61.24
MobileHumanPose	60.79

4.3.3 Visualization of prediction results

Apart from comparing the prediction accuracy of different models, the plots of predictions of 2D points and 3D points are showed to compare them in terms of visualization. The Figure 4.8 shows the prediction results from 2D-to-3D lifting model, including 2D ground-truth points, 3D ground-truth points and 3D predicted points. The Figure 4.9 shows the bounding box and 2D estimated keypoints of a single person and the Figure 4.10 shows the 3D estimated keypoints from the same models that produced the predicted 2D points. The Figure 4.11 shows the different prediction stages from the MobileHumanPose model, where 4.11(a) is the results of prediction of root/pelvis joints and 4.11(b) and 4.11(c) are the plot of 2D predicted and 3D predicted joints by MobileHumanPose model respectively. According to these figures, no obvious differences of 2D poses or 3D poses using different models was observed and the visualization results show that the human poses can be predicted correctly using all of models, tested in the section 4.3.2.

4.3.4 Model complexity

In addition to show the accuracy of predicted keypoints, the complexity of models should be considered when we deploy these models in real-time games or application. If visualization of predicted results of two models are similar, then the complexity of model is a key factor to select which model that can be utilized in a large software.

The complexity of model can be evaluated by number of parameters in a model and number of Floating-point Operations (FLOPs), and these floating operations can be an addition, subtraction, division and multiplication. The FLOPs of different layers (such as convolutional layer, pooling layer, etc.) were computed following the Equation 4.7.

$$FLOPs\,of\;Conv2D\;layer = 2 \times kernel\;size \times output\;size \tag{4.7}$$

$$FLOPs\,of\;Pooling\;layer = inputsize \tag{4.8}$$

$$FLOPs\,of\;Linear\;layer = 2 \times input\;size \times output\;size \tag{4.9}$$

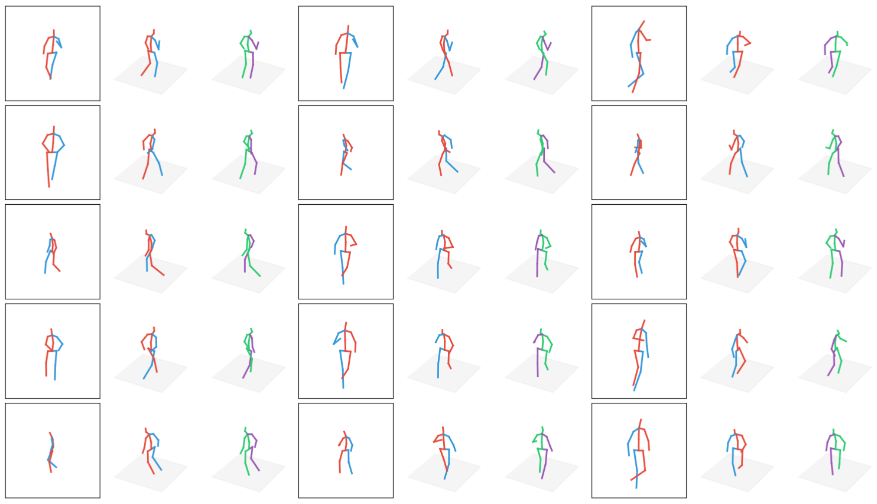


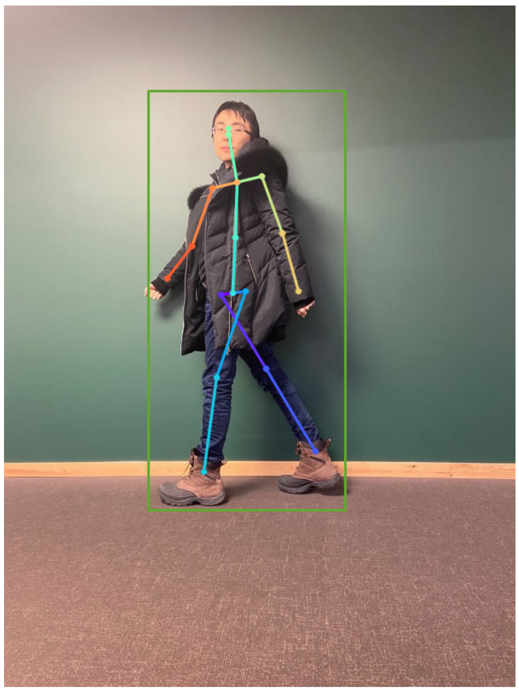
Figure 4.8 *The visualization of 3D pose predictions using 2D-3D lifting models*

In Equation 4.7, kernel, input, output is a tensor with the shape (B, C, W, H) and B denotes the batch size, that is the number of samples in a batch, c denotes the number of channel and W and H are width and height respectively. Therefore, the size of kernel, input and output is calculated by $B \times C \times W \times H$.

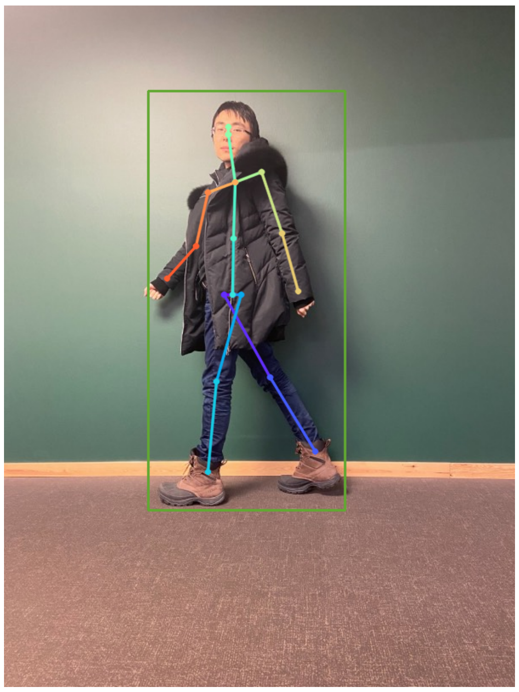
The Table 4.3 shows the input size, number of parameters and FLOPs of these tested models. From the table, for the 2D pose models, the Stacked Hourglass model has the most parameters and the Lite-HRNet model has the least parameters. Additionally, the 2D-3D lifting model is very small, but in the meantime, the prediction accuracy is the highest shown in the table 4.2, which means that the 3D pose estimation can be improved by simplifying the 2D pose models and increasing the prediction accuracy from the 2D pose models.

Table 4.3 *The number of parameters and FLOPs of different pose models*

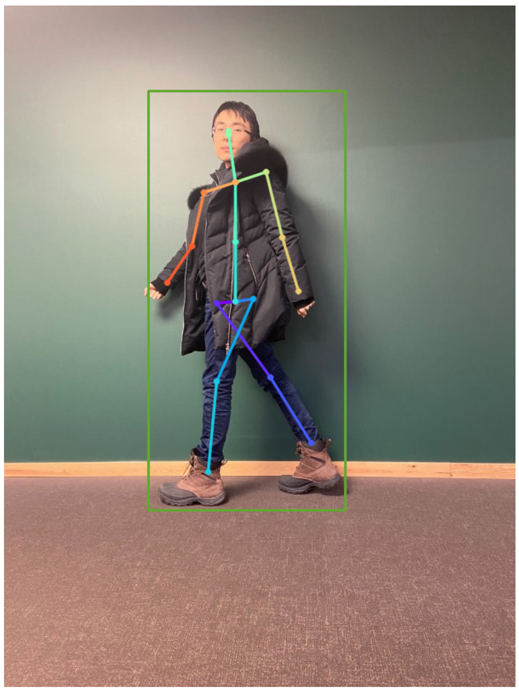
model	**input shape**	**number of parameters/M**	**GFLOPs**
Stacked Hourglass Network	(1,3,256,256)	94.85	28.67
HRFormer	(3,256,192)	7.75	2.99
Lite-HRNet	(3,256,192)	1.13	0.27
MobileNetV2	(3,256,192)	9.57	1.59
2D-3D Lifting	(34,1)	4.29	0.04
MobileHumanPose	(3,256,256)	34.34	14.32



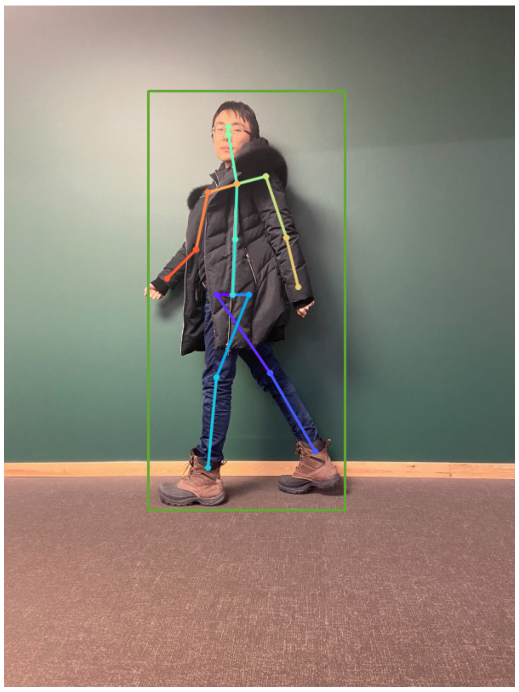
(a) Stacked Hourglass Network



(b) HRFormer

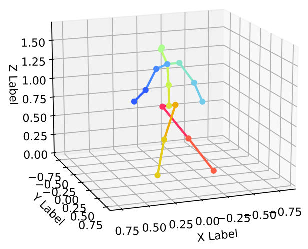


(c) Lite-HRNet

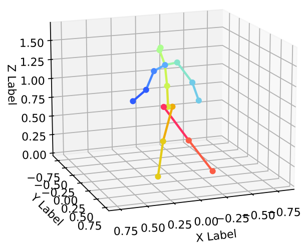


(d) MobileNetV2

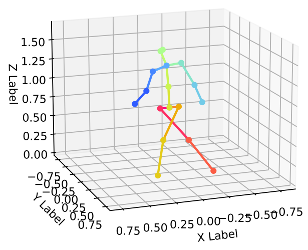
Figure 4.9 *Visualization of 2D pose predictions of different pose models*



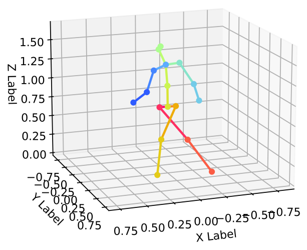
(a) Stacked Hourglass Network



(b) HRFormer



(c) Lite-HRNet



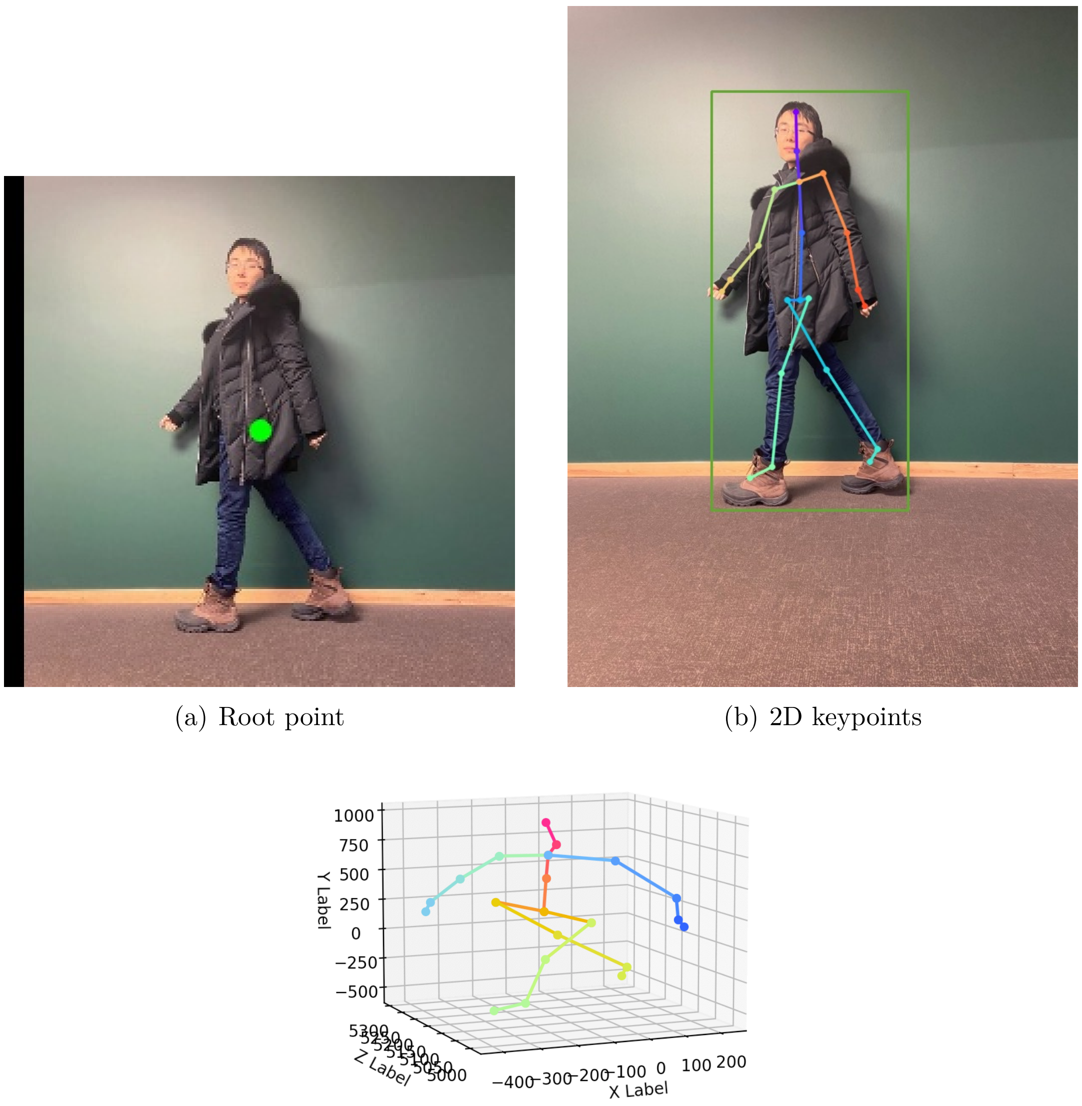
(d) MobileNetV2

Figure 4.10 *Visualization of 3D pose predictions of different pose models*

4.4 Testing pose models in real-time

Addition to testing accuracy of predicted keypoints from pose models on a dataset, the reality and fluency of character animation generated by predicted keypoints were tested in Unreal Engine 4 game engine. The reality of animation was observed manually, to test if animation matches user current pose and fluency of animation is represented by FPS (Frames Per Second). FPS was obtained by the equation $1/timedifferencebetweenlastframeandcurrentframe(seconds)$. In Unreal Engine 4 game engine, the time difference between two consecutive frames can be obtained from the parameter, named DeltaTime in a in-built function, void Tick(float DeltaTime), which is called every frame.

The whole pipeline to generate character animation is firstly reading frame from WebCam, next producing 3D coordinates of human keypoints and finally generating animation by setting locations of character's keypoints based on the predicted keypoints and finally the rotation of joints is calculated by inverse kinematics techniques provided by Unreal Engine 4 game engine. The detailed implementations of



(a) Root point (b) 2D keypoints

(c) 3D keypoints

Figure 4.11 *Visualization of predictions of MobileHumanPose model*

these stages are introduced in the section4.4.1, 4.4.2, 4.4.3 respectively.

The Table 4.4 shows the average FPS within around 3 minutes by using different models to predict human landmarks, and based on the result, the Lite-HRNet has a promising FPS, which can be used in games, where requiring pretty high FPS. But using other models, like HRFormer, MobileNetV2, MobileHumanPose, the program runs smoothly as well and no obvious lagging is observed.

However, the animation generated by real human pose is not very good, and the distinguished differences by using these models are not observed. Some animations are shown in the Figure 4.12. For the action, "turning the body", the animation seems quite real but for the action, like "lifting the arm" or "lowering the head",

***Table 4.4** The FPS of using different models in Unreal Engine 4 game engine*

model	**FPS**
Stacked Hourglass Network + Lifting	23
HRFormer + Lifting	48
Lite-HRNet + Lifting	55
MobileNetV2 + Lifting	54
MobileHumanPose	37

the animation of head is not similar as real actions, which means the prediction of depth (z axis) of the neck joint is not accurate.

4.4.1 Reading and processing frame from Webcam

I created a C++ class, called AWebCamReader, inheriting AActor class, and in this AWebCamReader class, I utilized OpenCV library to open webcam and reading and processing a single frame from the webcam, including, resizing the input frame, converting the color space of the input frame from BGR to RGB because the color space of frame from webcam is BGR but the input image of Pose estimation model is RGB.

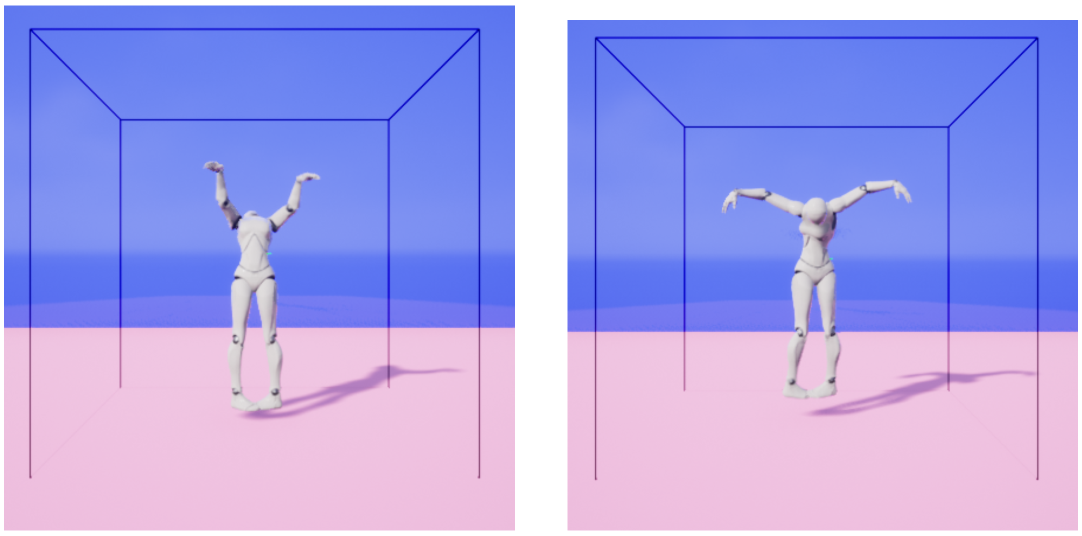
In the AWebCamReader class, mainly we need three variables to store the webcam stream, frame data and frame size.

```
cv::Mat m_videoFrame;
cv::VideoCapture m_videoStream;
cv::Size m_frameSize;
```

Additionally, there are some functions in the AWebCamReader class, to open the webcam, check if webcam is available, read frame data from webcam stream and process the frames, including resizing, converting color space and normalizing, such as BeginPlay(), UpdataFrame() and DoProcessing(), where the function BeginPlay() is overridden AActor::BeiginPlay() to initialize the class member variables at the beginning of the program. The implementation is shown in the Figure 4.13.

4.4.2 Model inference using input frame

To get 3D pose coordinates, I created a class called ATfLiteInference, which also inherits from AActor class. The functionality of this class is to acquire the instance of AWebCamReader class in the main program, get output from model and then store the 3D pose points. When initializing the object, we should define the model to use, and hyperparameters of the model, like the size of input tensor, the amount of the pose points we want to get from the output. At the beginning of the program, I firstly loaded the model, and initialized the Tensorflow Lite interpreter, which is used


(a) Lifting arms (b) Lowering head

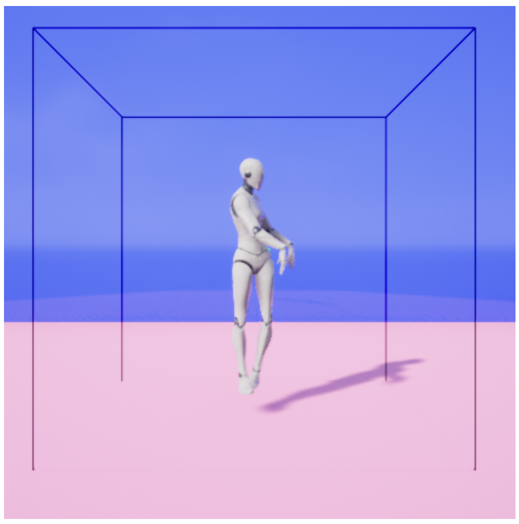
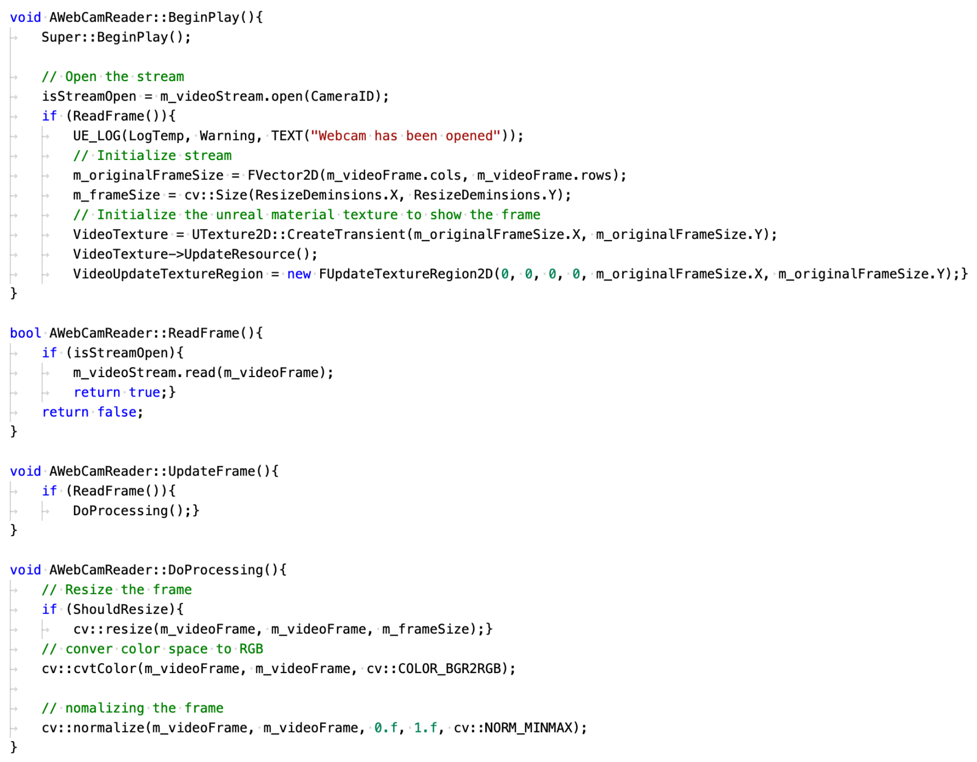

(c) Turning body

Figure 4.12 *The animation generated by pose models*

for inference. In every iterator, the functions UpdateFrame(), FillInputBuffers() and InvokeInterpreter() are called one by one, where I got the frame data by calling the function UpdateFrame(), and filled the frame data into to the input tensor of model by calling FillInputBuffers() and made an inference by calling InvokeInterpreter(). The implementation is shown in the Figure 4.14.

Generally, the outputs, number of joint points are different if we use different models and the correspondences between skeletal name and output index are different. For instance, in model1, the output is x, y, z, score of 17 joints, and the 15th point represents the skeletal joint of left feet. While in the model2, there are 32 joints and 15th point represents the skeletal joint of left hand. Thus, addition to outputting 3D points, we need a variable to store the pairs of names of skeletal joint



```
void AWebCamReader::BeginPlay(){
    Super::BeginPlay();

    // Open the stream
    isStreamOpen = m_videoStream.open(CameraID);
    if (ReadFrame()){
        UE_LOG(LogTemp, Warning, TEXT("Webcam has been opened"));
        // Initialize stream
        m_originalFrameSize = FVector2D(m_videoFrame.cols, m_videoFrame.rows);
        m_frameSize = cv::Size(ResizeDeminsions.X, ResizeDeminsions.Y);
        // Initialize the unreal material texture to show the frame
        VideoTexture = UTexture2D::CreateTransient(m_originalFrameSize.X, m_originalFrameSize.Y);
        VideoTexture->UpdateResource();
        VideoUpdateTextureRegion = new FUpdateTextureRegion2D(0, 0, 0, 0, m_originalFrameSize.X, m_originalFrameSize.Y);}
}

bool AWebCamReader::ReadFrame(){
    if (isStreamOpen){
        m_videoStream.read(m_videoFrame);
        return true;}
    return false;
}

void AWebCamReader::UpdateFrame(){
    if (ReadFrame()){
        DoProcessing();}
}

void AWebCamReader::DoProcessing(){
    // Resize the frame
    if (ShouldResize){
        cv::resize(m_videoFrame, m_videoFrame, m_frameSize);}
    // conver color space to RGB
    cv::cvtColor(m_videoFrame, m_videoFrame, cv::COLOR_BGR2RGB);

    // nomalizing the frame
    cv::normalize(m_videoFrame, m_videoFrame, 0.f, 1.f, cv::NORM_MINMAX);
}
```

Figure 4.13 *C++ implementation for reading frames from WebCam*

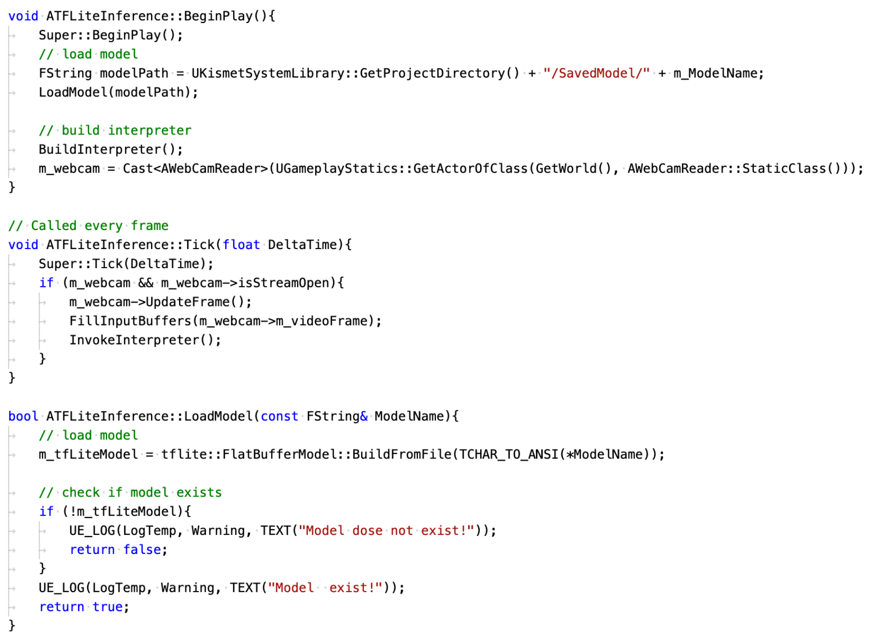
and landmark IDs in this class, because they are related to the which model to use.

4.4.3 Animating character based on 3D joint points

The animation of the character is controlled by the UNREAL class, called UPhysicsHandleComponent, where we set the locations of skeletal joints and UNREAL will use IK techniques to calculate the rotations of the skeletal joints and finally we can see the animation of the character by seeing the rotation of each skeletal joints of the character. In the function InitialBone(), I initialize the instance of the class UPhysicsHandleComponent for each skeletal joint and in the function UpdateBone(), I acquire the 3D points from the model object, and the set the new location to each skeletal joints if the possibility of visibility of the points is larger than a determined threshold. The implementation is shown in the Figure 4.15.

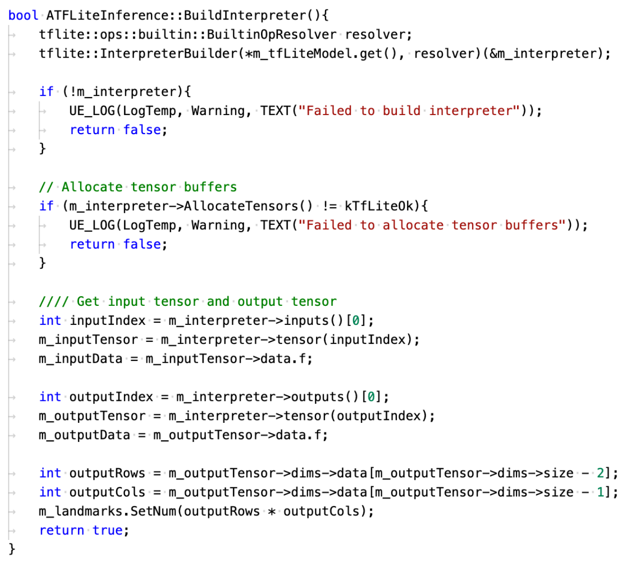
4.5 Discussion

This thesis estimates 3D human pose using two approaches, the end-to-end model and 2D-to-3D lifting method. The output of end-to-end model is 3D skeletal locations while 2D-to-3D lifting method has two stages that feeding input images to 2D pose model to obtain 2D skeletal locations and then feeding them to 2D-to-3D lifting model to get 3D skeletal locations. There are four 2D models with the same



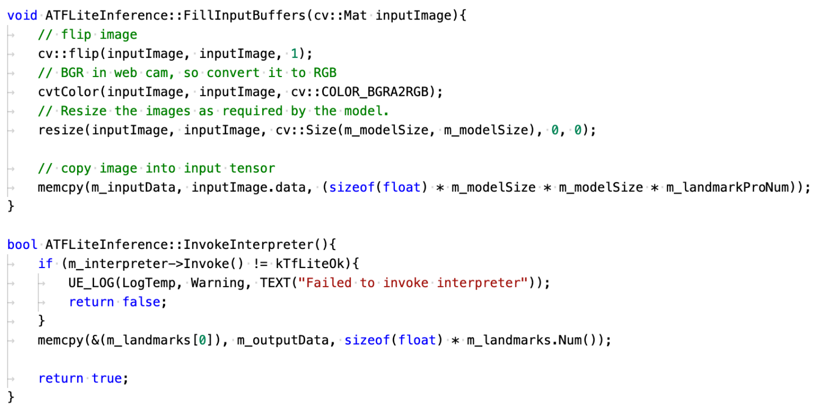
```
void ATFLiteInference::BeginPlay(){
    Super::BeginPlay();
    // load model
    FString modelPath = UKismetSystemLibrary::GetProjectDirectory() + "/SavedModel/" + m_ModelName;
    LoadModel(modelPath);

    // build interpreter
    BuildInterpreter();
    m_webcam = Cast<AWebCamReader>(UGameplayStatics::GetActorOfClass(GetWorld(), AWebCamReader::StaticClass()));
}

// Called every frame
void ATFLiteInference::Tick(float DeltaTime){
    Super::Tick(DeltaTime);
    if (m_webcam && m_webcam->isStreamOpen){
        m_webcam->UpdateFrame();
        FillInputBuffers(m_webcam->m_videoFrame);
        InvokeInterpreter();
    }
}

bool ATFLiteInference::LoadModel(const FString& ModelName){
    // load model
    m_tfLiteModel = tflite::FlatBufferModel::BuildFromFile(TCHAR_TO_ANSI(*ModelName));

    // check if model exists
    if (!m_tfLiteModel){
        UE_LOG(LogTemp, Warning, TEXT("Model dose not exist!"));
        return false;
    }
    UE_LOG(LogTemp, Warning, TEXT("Model  exist!"));
    return true;
}
```

(a) Part1 of functions in the implementation



```
bool ATFLiteInference::BuildInterpreter(){
    tflite::ops::builtin::BuiltinOpResolver resolver;
    tflite::InterpreterBuilder(*m_tfLiteModel.get(), resolver)(&m_interpreter);

    if (!m_interpreter){
        UE_LOG(LogTemp, Warning, TEXT("Failed to build interpreter"));
        return false;
    }

    // Allocate tensor buffers
    if (m_interpreter->AllocateTensors() != kTfLiteOk){
        UE_LOG(LogTemp, Warning, TEXT("Failed to allocate tensor buffers"));
        return false;
    }

    //// Get input tensor and output tensor
    int inputIndex = m_interpreter->inputs()[0];
    m_inputTensor = m_interpreter->tensor(inputIndex);
    m_inputData = m_inputTensor->data.f;

    int outputIndex = m_interpreter->outputs()[0];
    m_outputTensor = m_interpreter->tensor(outputIndex);
    m_outputData = m_outputTensor->data.f;

    int outputRows = m_outputTensor->dims->data[m_outputTensor->dims->size - 2];
    int outputCols = m_outputTensor->dims->data[m_outputTensor->dims->size - 1];
    m_landmarks.SetNum(outputRows * outputCols);
    return true;
}
```

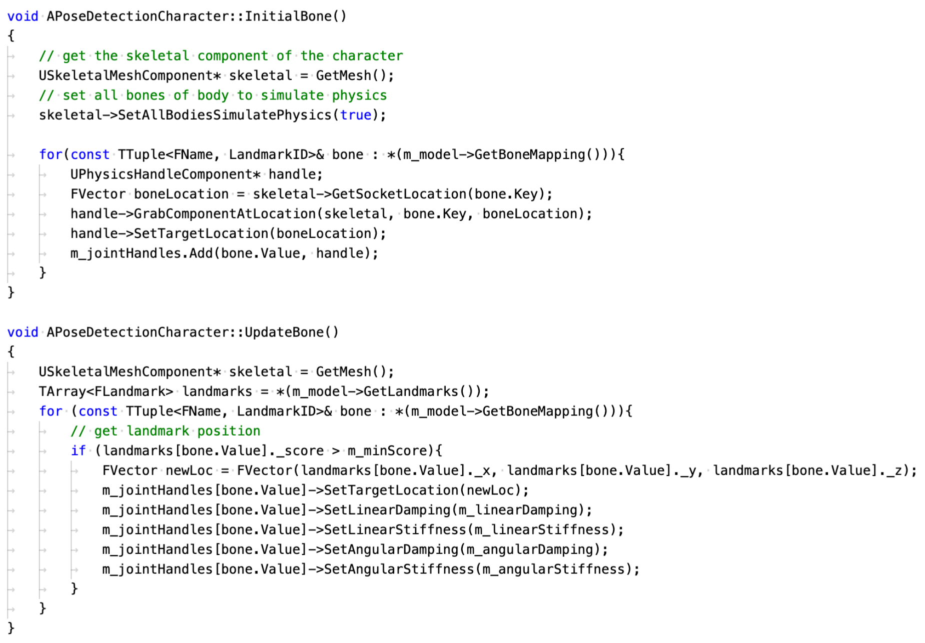
(b) Part2 of functions in the implementation



```
void ATFLiteInference::FillInputBuffers(cv::Mat inputImage){
    // flip image
    cv::flip(inputImage, inputImage, 1);
    // BGR in web cam, so convert it to RGB
    cvtColor(inputImage, inputImage, cv::COLOR_BGRA2RGB);
    // Resize the images as required by the model.
    resize(inputImage, inputImage, cv::Size(m_modelSize, m_modelSize), 0, 0);

    // copy image into input tensor
    memcpy(m_inputData, inputImage.data, (sizeof(float) * m_modelSize * m_modelSize * m_landmarkProNum));
}

bool ATFLiteInference::InvokeInterpreter(){
    if (m_interpreter->Invoke() != kTfLiteOk){
        UE_LOG(LogTemp, Warning, TEXT("Failed to invoke interpreter"));
        return false;
    }
    memcpy(&(m_landmarks[0]), m_outputData, sizeof(float) * m_landmarks.Num());

    return true;
}
```

(c) Part3 of functions in the implementation

Figure 4.14 *C++ implementation for generating 3D human landmarks*



```cpp
void APoseDetectionCharacter::InitialBone()
{
    // get the skeletal component of the character
    USkeletalMeshComponent* skeletal = GetMesh();
    // set all bones of body to simulate physics
    skeletal->SetAllBodiesSimulatePhysics(true);

    for(const TTuple<FName, LandmarkID>& bone : *(m_model->GetBoneMapping())){
        UPhysicsHandleComponent* handle;
        FVector boneLocation = skeletal->GetSocketLocation(bone.Key);
        handle->GrabComponentAtLocation(skeletal, bone.Key, boneLocation);
        handle->SetTargetLocation(boneLocation);
        m_jointHandles.Add(bone.Value, handle);
    }
}

void APoseDetectionCharacter::UpdateBone()
{
    USkeletalMeshComponent* skeletal = GetMesh();
    TArray<FLandmark> landmarks = *(m_model->GetLandmarks());
    for (const TTuple<FName, LandmarkID>& bone : *(m_model->GetBoneMapping())){
        // get landmark position
        if (landmarks[bone.Value]._score > m_minScore){
            FVector newLoc = FVector(landmarks[bone.Value]._x, landmarks[bone.Value]._y, landmarks[bone.Value]._z);
            m_jointHandles[bone.Value]->SetTargetLocation(newLoc);
            m_jointHandles[bone.Value]->SetLinearDamping(m_linearDamping);
            m_jointHandles[bone.Value]->SetLinearStiffness(m_linearStiffness);
            m_jointHandles[bone.Value]->SetAngularDamping(m_angularDamping);
            m_jointHandles[bone.Value]->SetAngularStiffness(m_angularStiffness);
        }
    }
}
```

Figure 4.15 *C++ implementation for animating characters*

2D-to-3D lifting model and one 3D model selected for evaluation and comparison according to the relatively better performances provided by the 3D pose estimation survey (Zheng et al. 2020). Furthermore, this thesis created an application to generate real-time animations using estimated 3D skeletal animations in Unreal Engine 4 game engine.

Based on the subjective observation, the generated animations have some obvious flaws. Firstly, the character animation looks shaking even though the real pose is not moving, which is because the estimated locations are fluctuated slightly and these shaky data leads to the shaking animations. Secondly, the estimation for Z-axis or depth is not accurate enough. For example, when moving horizontally or vertically, the corresponding animations shows character moves correctly, while when moving forward and back (along the Z-axis), the animations are distorted, especially the character's head is rose and lowered frequently but the head of real pose is erect. Moreover, the knee joints and elbow joints always turn inward. Actually, from the evaluation results, the 2D models show the less errors compared to 3D models.

Therefore, in the next step, the improvement of generating real-time animations may involves solving shaking problem and enhance accuracy for Z-axis of 3D skeletal locations. Shaking problem may get solved by detecting movement of human. The inaccurate estimation of depth information is a important problem mentioned by many research work and the main reason is that the ground truth of Z-axis is already imprecise because collection of 3D data is more difficult than obtaining 2D data. Therefore, obtaining accurate 3D data still needs to be studied.

5 Conclusion

This thesis researches the advanced deep neural networks for human pose estimation and based on the evaluation results provided by the corresponding project papers, the four optimal neural network architectures(Stacked Hourglass, HRFormer, LiteHRNet, MobileNetV2) for 2D human pose estimation and MobileHumanPose for 3D human pose estimation are selected for training and testing in this thesis. The purpose of this thesis is utilizing predicted 3D landmarks to generate real-time animations, thus 2D to 3D pose lifting neural network are applied following the prediction from 2D human pose. This thesis first trained four 2D human pose estimation models on the COCO dataset and trained 2D to 3D pose lifting neural network on the Human3.6M dataset. Then the combination of 2D human pose model and 3D pose lifting model can be used to generate 3D coordinates of human keypoints from a single image. In order to compare the human pose estimation models on a common evaluation standards, this thesis tested MPJPE score of these models on the same dataset, Human3.6M. And the qualities of predicted keypoints from different models are visualized and analyzed. Furthermore, the model complexity, such as FLOPS, number of model parameters are showed in this thesis. Finally, an application is built for generating character's animation in real time based on user's actions and the quality of generated animations are analyzed.

The quantitative evaluation of 3D human pose estimation models is based on the MPJPE score, which denotes the distance between predicted joints and ground-truth. The MPJPE scores of stacked hourglass + lifting, HRFormer + lifting, LiteHRNet + lifting, MobileNetV2 + lifting, MobileHumanPose are 54.58mm, 58.70mm, 58.72mm, 61.24mm and 60.79mm respectively, which illustrates stacked hourglass model can produce the most accurate predicted keypoints among these models, while the LiteHRNet has the highest inference speed. However, from the visualization of inference of self-taken photos, no obvious differences of results from these models are shown and this situation is same when comparing the animations generated by these different models in Unreal Engine 4 game engine. For the qualitative evaluation of animations based on the predicted human pose keypoints, all of these models produce inaccurate depth value, which is reflected by the animation is worse when rotating or moving z-axis.

References

Bulat, Adrian et al. (2019). *Improved training of binary networks for human pose estimation and image recognition.* arXiv: 1904.05868 [cs.CV].

Choi, Sangbum, Seokeon Choi, and Changick Kim (June 2021). "MobileHumanPose: Toward Real-Time 3D Human Pose Estimation in Mobile Devices". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pp. 2328–2338.

Chollet, François (2016). "Xception: Deep Learning with Depthwise Separable Convolutions". In: *CoRR* abs/1610.02357. arXiv: 1610.02357. URL: http://arxiv.org/abs/1610.02357.

Dosovitskiy, Alexey et al. (2020). "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *CoRR* abs/2010.11929. arXiv: 2010.11929. URL: https://arxiv.org/abs/2010.11929.

Fukushima, Kunihiko and Sei Miyake (1982). "Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position". In: *Pattern Recognition* 15.6, pp. 455–469. ISSN: 0031-3203. DOI: https://doi.org/10.1016/0031-3203(82)90024-3. URL: https://www.sciencedirect.com/science/article/pii/0031320382900243.

He, Kaiming, Georgia Gkioxari, et al. (2018). *Mask R-CNN.* arXiv: 1703.06870 [cs.CV].

He, Kaiming, Xiangyu Zhang, et al. (2015). *Deep Residual Learning for Image Recognition.* arXiv: 1512.03385 [cs.CV].

Hinton, Geoffrey et al. (2012). "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups". In: *IEEE Signal Processing Magazine* 29.6, pp. 82–97. DOI: 10.1109/MSP.2012.2205597.

Hoiem, Derek, Yodsawalai Chodpathumwan, and Qieyun Dai (Oct. 2012). "Diagnosing Error in Object Detectors". In: pp. 340–353. ISBN: 978-3-642-33711-6. DOI: 10.1007/978-3-642-33712-3_25.

Ioffe, Sergey and Christian Szegedy (2015). *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.* arXiv: 1502.03167 [cs.LG].

Ionescu, Catalin et al. (2014). "Human3.6M: Large Scale Datasets and Predictive Methods for 3D Human Sensing in Natural Environments". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.7, pp. 1325–1339. DOI: 10.1109/TPAMI.2013.248.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems.* Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

LeCun, Y. et al. (1989). “Backpropagation Applied to Handwritten Zip Code Recognition”. In: *Neural Computation* 1, pp. 541–551.

Lecun, Y. et al. (1998). “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11, pp. 2278–2324. DOI: 10.1109/5.726791.

Lemaréchal, Claude (2010). *Cauchy and the Gradient Method*. https://www.math.uni-bielefeld.de/documenta/vol-ismp/40_lemarechal-claude.pdf, Last accessed on 2022-06-23.

Lin, Tsung-Yi, Piotr Dollár, et al. (2017). *Feature Pyramid Networks for Object Detection*. arXiv: 1612.03144 [cs.CV].

Lin, Tsung-Yi, Michael Maire, Serge Belongie, et al. (2015). *Microsoft COCO: Common Objects in Context*. arXiv: 1405.0312 [cs.CV].

Lin, Tsung-Yi, Michael Maire, Serge J. Belongie, et al. (2014). “Microsoft COCO: Common Objects in Context”. In: *CoRR* abs/1405.0312. arXiv: 1405.0312. URL: http://arxiv.org/abs/1405.0312.

Ma, Ningning et al. (2018). “ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design”. In: *CoRR* abs/1807.11164. arXiv: 1807.11164. URL: http://arxiv.org/abs/1807.11164.

Martinez, Julieta et al. (2017). *A simple yet effective baseline for 3d human pose estimation*. arXiv: 1705.03098 [cs.CV].

Mcculloch, Warren and Walter Pitts (1943). “A Logical Calculus of Ideas Immanent in Nervous Activity”. In: *Bulletin of Mathematical Biophysics* 5, pp. 127–147.

Moon, Gyeongsik, Ju Yong Chang, and Kyoung Mu Lee (2019). *Camera Distance-aware Top-down Approach for 3D Multi-person Pose Estimation from a Single RGB Image*. arXiv: 1907.11346 [cs.CV].

Newell, Alejandro, Kaiyu Yang, and Jia Deng (2016). “Stacked Hourglass Networks for Human Pose Estimation”. In: *CoRR* abs/1603.06937. arXiv: 1603.06937. URL: http://arxiv.org/abs/1603.06937.

NVIDIA (2022a). *CUDA Toolkit*. https://developer.nvidia.com/cuda-toolkit, Last accessed on 2022-03-25.

— (2022b). *NVIDIA cuDNN*. https://developer.nvidia.com/cudnn, Last accessed on 2022-03-25.

— (2022c). *NVIDIA website*. https://www.nvidia.com/en-us/, Last accessed on 2022-06-23.

Redmon, Joseph and Ali Farhadi (2018). *YOLOv3: An Incremental Improvement*. DOI: 10.48550/ARXIV.1804.02767. URL: https://arxiv.org/abs/1804.02767.

Robbins, Herbert E. (2007). “A Stochastic Approximation Method”. In: *Annals of Mathematical Statistics* 22, pp. 400–407.

Rosenblatt, F. (1958). “The perceptron: A probabilistic model for information storage and organization in the brain.” In: *Psychological Review* 65.6, pp. 386–408. ISSN: 0033-295X. DOI: 10.1037/h0042519. URL: http://dx.doi.org/10.1037/h0042519.

Rumelhart, D. E., J. L. McClelland, and PDP Research Group, eds. (1986). *Parallel Distributed Processing. Volume 1: Foundations*. Cambridge, MA: MIT Press.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (1988). “Learning Representations by Back-Propagating Errors”. In: *Neurocomputing: Foundations of Research*. Cambridge, MA, USA: MIT Press, pp. 696–699. ISBN: 0262010976.

Sandler, Mark et al. (2018). “Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation”. In: *CoRR* abs/1801.04381. arXiv: 1801.04381. URL: http://arxiv.org/abs/1801.04381.

Sun, Xiao et al. (2018). *Integral Human Pose Regression*. arXiv: 1711.08229 [cs.CV].

Tompson, Jonathan et al. (2014). “Joint Training of a Convolutional Network and a Graphical Model for Human Pose Estimation”. In: *CoRR* abs/1406.2984. arXiv: 1406.2984. URL: http://arxiv.org/abs/1406.2984.

Unreal Engine 4.27 documentation (2021). *Skeletal Mesh Animation System*. https://docs.unrealengine.com/4.27/en-US/AnimatingObjects/SkeletalMeshAnimation/, Last accessed on 2022-04-06.

Vinyals, Oriol et al. (2017). “StarCraft II: A New Challenge for Reinforcement Learning”. In: *CoRR* abs/1708.04782. arXiv: 1708.04782. URL: http://arxiv.org/abs/1708.04782.

Wang, Jingdong et al. (2019). “Deep High-Resolution Representation Learning for Visual Recognition”. In: *CoRR* abs/1908.07919. arXiv: 1908.07919. URL: http://arxiv.org/abs/1908.07919.

Yu, Changqian et al. (2021). “Lite-HRNet: A Lightweight High-Resolution Network”. In: *CoRR* abs/2104.06403. arXiv: 2104.06403. URL: https://arxiv.org/abs/2104.06403.

Yuan, Yuhui et al. (2021). “HRFormer: High-Resolution Transformer for Dense Prediction”. In: *CoRR* abs/2110.09408. arXiv: 2110.09408. URL: https://arxiv.org/abs/2110.09408.

Zheng, Ce et al. (2020). *Deep Learning-Based Human Pose Estimation: A Survey*. DOI: 10.48550/ARXIV.2012.13392. URL: https://arxiv.org/abs/2012.13392.