Human Pose Estimation for Video Game Control

Rakesh Johny, Tom Li, Aditya Narayanan, Albert Xia

Abstract

Current methods of interacting with computers are flawed in a couple of key ways: they fail to map physically-intuitive motions to their computer control counterparts, and they rely heavily on a user's fine-motor skills, which are heavily impacted by factors such as muscle coordination disabilities and old age. In this paper, we propose a system of computer control through gesture tracking via real-time human pose estimation on an embedded device, which is capable of addresssing these issues, by mapping general physically-intuitive motions into computer control. Furthermore, our device boasts low latency in the human-computer interaction, and is minimally intrustive to the computer being controlled. We test the viability of our system as a computer-control device by playing two video games using only gestures.

1. Introduction

The human-computer interfaces behind many modern video games use either handheld joystick controllers or keyboard and mouse input, which are examples of typical human input device for computers, monitors, televisions, etc. There are two major drawbacks to traditional human input systems that are often overlooked: the input needed to achieve a certain effect has little to no correspondence to a physically-intuitive set of motions, and more importantly, inputs are heavily dependent on the user's fine motor skills. Fine motor skills used to interact with modern technology are heavily affected by factors such as muscular coordination disorders and old age. Computer vision, particularly the processes of human pose estimation and motion tracking may allow us to create human-computer interfaces that link computer control to physically-intuitive human motion inputs with little dependence on fine motor ability or additional physical input devices. We propose a vision-based computer-input system that maps gestures and motion of the user's body into computer input. We test the system's effectiveness with the playability of two simple video games as metrics. In order to demonstrate the versatility of our phased-approach to gesture-based control, we select a car-racing game and the Google Chrome Dino Racer game as test video games. Such a system, when expanded to replicating general keyboard + mouse control, would be instrumental in improving the accessibility of modern technology, with implementations far beyond the scope of computer games.

2. Related Work

2.1. Human Pose Estimation

Some of the largest components of our system fall into the category of so called Human Pose Estimation or the ability to properly:

- Identify users as sources of input, from a video stream
- Segment the user's body to isolate relevant portions of data
- Compute the location, in image coordinates, of keypoints on the user's body.

The area of Human Pose Estimation is widely studied. There are a wide array of robust, optimized solutions for gathering 2D image coordinates of keypoints from images of users.

2.1.1 Openpose

One popular solution to realtime human pose estimation, derived from [2], uses Part Affinity Fields (PAFs) to learn to assosciate body parts with individuals in the image, and open-source code for the so called openpose library shows promising results for real-time human body segmentation.

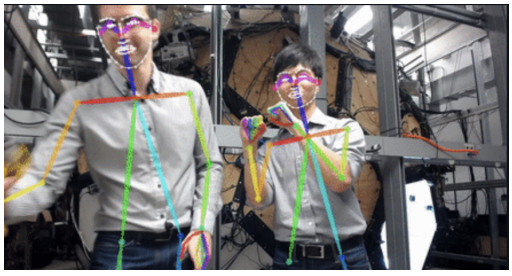

Figure 1. A demonstration of openpose performing human body segmentation.

2.1.2 MoveNet and PoseNet

PoseNet and its more recent counterpart MoveNet are fully convolutional neural network pose estimators built on existing ResNet or MobileNet backbones [6]. Benchmarks of PoseNet and MoveNet show that it is viable to run either estimator in real time on embedded devices with appropriate model quantizations [1], with PoseNet being slightly faster when compared to MoveNet at identical levels of quantization. In comparison's with Openpose, PoseNet/MoveNet are shown to perform slightly better on instance segmentation of the person class from the COCO dataset.

2.2. Gesture Identification/Motion Tracking

Openpose and MoveNet/PoseNet give us promising ways to identify locations of human body features (hands, arms, etc.) in a video stream. However, this is not enough to categorize a user's motion as a particular command to a computer system. To be able to extract feature motion over time as a useful input to our car racing video game, we will have to implement motion tracking over a video stream.

A number of guides exist for implementing feature tracking over video. [7] and [5] demonstrate simple object tracking across video frames, and [5] demonstrates a system shown to be accurate with motion of hands. We drew on [7] and [5] in developing a tracking algorithm for the identification of a human user ducking or jumping from a keypoint cloud.

3. Methodology

Our human computer interface consists of three major steps: Pose Estimation, or feature location, Feature Tracking and classification of motions, and video game interfacing. In addition, we adopt a multi-phased approach to our human computer interface. Phase 1 consists of a robust and general human pose estimator and keypoint detector. The work in Phase 1 is task/game agnostic, as we implement identical versions of this phase for the car-racer game as well as the chrome dinosaur game. Phase 2 is a game-specific keypoint classifier, where the output from Phase 1 is classified into a number of computer commands in a way dependent on the physical motions being replicated in the game as well as the computer commands the game is traditionally played with.

The motivation behind such a two phase approach rather than building a classifier that directly maps input video streams to computer commands is to improve the versatility of our human-computer interaction system. A fully general Phase 1 implemented independent of the game in question allows the system to be more easily adapted to any particular set of computer commands. We demonstrate the versatility of this system through the implementation of our human-computer interface for two distinct video games that utilize identical Phase 1 code.

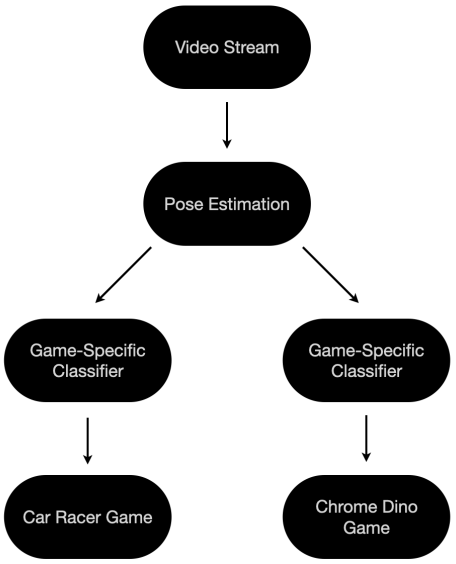

Figure 2. Data flow within our human-computer interface system. Note the use of a universal pose estimator and game-specific classifiers.

3.1. Testing Pose Estimation Methods

Before we develop phase 1 of our human-computer interface, we test a number of proposed solutions for human pose estimation to determine which ones are best-suited to our needs. The main criteria we require for our human pose estimator are:

- Real time operation. The need for this criterion is obvious, as we require low latency classification in order to mimic low-latency commmands for video games. Essentially, the user should feel as if the delay between executing a physical command and the video game reflecting such a command is instantaneous.
- The system should be minimally intrusive on the computer used to play the game. This essentially means that we cannot use computing resources so significant that usability of the computer itself is diminished.

3.1.1 Openpose Python API

The openpose package offers a Python API for implementing inference over video feeds using one of the openpose models. We began testing the openpose Python API for our task, but found that installing the API resulted in dependency issues accross multiple team members' computers and decided to instead test the openpose pretrained models implemented independently of the openpose python API.

3.1.2 Openpose model implemented in OpenCV

[4] offers a method to perform inference on CPU/GPU using pretrained versions of the openpose hand/body pose estimators. For inference with the openpose pretrained model through OpenCV, we observed only 1-2 FPS on a 2.3 GHz Intel i9 CPU. We expect that running the same model with CUDA acceleration would result in true real-time performance, but as one of our design constraints for our system, we imposed that we cannot use significant GPU resources in order to maintain minimal intrusiveness onto the gaming machine.

3.1.3 PoseNet and the Jetson Nano

Due to the poor performance of the Openpose models, we began to look at different methods for real-time pose estimation. One option that was promising was to completely offload the task of human pose estimation to an edge-device to reduce the computational burden on the device used to play the games. Such an embedded device would ideally be compact yet powerful enough to run quantized versions of pose estimators in real time. We found the NVIDIA Jetson Nano to be an ideal candidate for such an embedded device.

With a 128-core NVIDIA GPU on a single-board footprint, the Jetson Nano is a popular choice for deep model inference on embedded computers. Perhaps more important than the hardware itself, however, is the specific Tensor RT optimized models available for use on the Jetson Nano. Through a number of optimization techniques such as model quantization, unuzed output elimination and horizontal layer fusion/layer aggregation, the Tensorflow with Tensor RT (TF-TRT) backend is able to greatly improve inference time on the Jetson Nano. TF-TRT is able to output optimized models from Open Neural Network Exchange Format (ONXX) and for popular tasks such as human pose estimation, ONXX model formats are readily available from NVIDIA Developers.

Using a variant of PoseNet built on a ResNet-18 backbone, we observe 15-17 Frames Per Second (FPS) on the Jetson Nano. The so called Pose-ResNet-18-body outputs 17 detected keypoints in image coordinate format. 15-17 FPS performance is smooth enough keypoint detection to allow us to play games in real time.

3.2. Pose Estimation

After testing the variety of available solutions for human pose estimation above, we identify that running TRT optimized PoseNet through the Jetson Nano is the best solution for our application.

To build the first phase of our human-computer interaction device, we use the jetson-inference and jetson-utils python packages ([3]) to build a human pose estimator in Python. The output of this step is a collection of 2D keypoints, correpsonding to image coordinates of the locations of the 17 body parts detected by PoseNet.

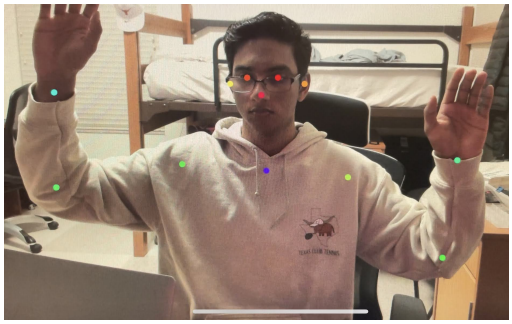

Figure 3. An example of the keypoints detected from a frame of video by Pose-ResNet-18

3.3. Feature Tracking and Classification

The second major phase of the human-computer interaction device are the keypoint classifiers, which receive 2d image coordinates keypoint detections from the pose estimator, and classify collections of keypoints into computer commands in a game-specific way. In a fully generalized model of our human-computer interaction device, the keypoint "decoder" built in this step need not even be a classifier. For more complicated continuous inputs to video games, this can be a more complicated regression problem as we would seek to estimate continuous output based on the locations of human pose keypoints.

We construct two distinct keypoint classifiers, one for each video game we demonstrate our system on. We construct one classifier to derive a steering command from pose keypoints, and one to derive jumping/ducking information from an identical set of keypoints.

3.3.1 Feature Classification For Steering

The first of two example video games we play as demonstrations of our human-computer interface is a simple car-racing game, in which the player's vehicle must dodge obstacles by steering left and right. The video game accepts controls as either left arrow key or right arrow key, so our keypoint classifier for this game seeks to classify collections of keypoints derived from our human pose estimator into either steer left or steer right commands.

To maintain a physical intuition to the gestures that are mapped to either left arrow key or right arrow key, we re-

quire a user to steer the car with an imaginary steering wheel on the video stream. From each frame of the user's hands on an imaginary wheel, we extract classifications for the game control as follows:

1. Assert that both the left wrist and right wrist of the user on the imaginary wheel are present in the frame. If one or more wrists are not present in the frame, skip the remaining steps and output no steering command.
2. Compute the vector connecting the left wrist of the user to the right wrist of the user.
3. Compute the angle to the horizontal of the vector computed in step 2.
4. Threshold on the angle to the horizontal with an experimentally determined cutoff threshold.

Let the location (in image coordinates) of the user's left wrist be (x, y) and the location of the user's right wrist be (x', y'). The displacement vector connecting the two wrists is given by:

$$d = (x - x', y - y')$$

The angle to the horizontal is then given by:

$$\theta = \arctan \frac{y - y'}{x - x'}$$

After some experimentation with the system, we found that a reasonable threshold for the steering angle is 0.2 radians. Our classifications for steering angle are then derived as:

- $\theta > 0.2$: Steer left
- $-0.2 < \theta < 0.2$: No command
- $-0.2 < \theta$: Steer Right

3.3.2 Feature Classification For Jumping/Ducking

The second example video game we demonstrate our human-computer interface system on is the Google Chrome Dinosaur Game. The game accepts jumping and ducking commands in the form of up arrow key and down arrow key; however, our classifier is a little more complex than for the first example video game, as ducking and jumping require motion tracking over video to properly classify from human motion. That is, you cannot determine from a single frame of video whether or not the user is ducking or jumping.

In order to deisgn an appropriate classifier for this task, we first establish a subset of the keypoints as a more descriptive feature to track. In the case of the car steering example, this was done by reducing the set of keypoints to simply the left wrist and right wrist. For classification of jumping and ducking, this is accomplished by computing a "mean face" or the midpoint of a number of selected keypoints. For this application, we select the two eyes, two ears, and the nose, and compute the midpoint of these keypoints as our augmented feature to track in order to classify whether a user is jumping or ducking. The motivation for establishing this feature is to improve the robustness of our classification. Were we to simply track a single keypoint such as the nose of the user, our system would not be stable against random errors when the nose may not be properly classified or may not be classified at all from the phase 1 human pose estimation. Running relatively small versions of PoseNet, these types of errors are fairly common. Establishing the midpoint of the face as a feature to track allows us to track an equally useful but more robust feature.

After establishing the midpoint feature, we obtain graphs of the vertical position and derivative of vertical position of the midpoint feature over time as a user performs jumping and ducking motions in order to develop a game-specific classifier for this task. A few methods we considered and our analysis of them are given below.

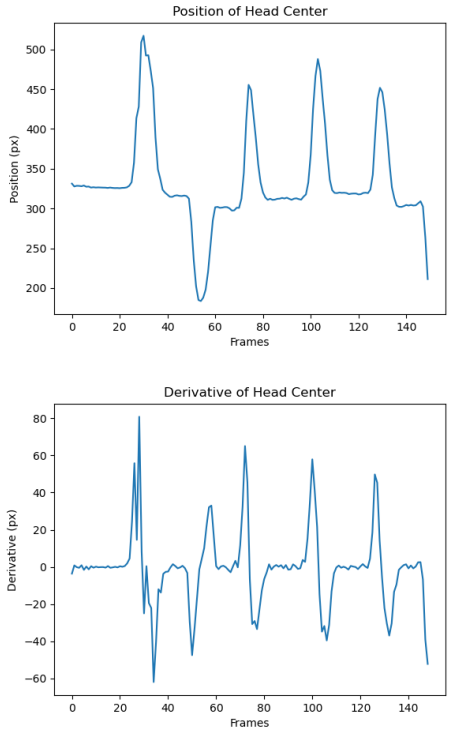

Figure 4. Top: y-position of face midpoint over time. Bottom: derivative of y-position of face midpoint over time.

1. **Thresholding on Position**: The most simple method we could employ to classify the movement of the midpoint face into either jumping or ducking would be to threshold on the vertical position in the frame of the midpoint of the users's face. However, this system is not invariant to small motions of the user between jumps and ducks. For example, if the user moves closer to or farther from the camera, this method breaks down.

2. **Threshold on Derivative**: One method we considered would be to simply threshold on the derivative of the y-position to discriminate between periods of slow/no motion, and those where the user rapidly moves up or down as one would during a jump or a duck. However, as we can see from the graphs of the derivative over time, both jumps and ducks have periods of high derivative in both the positive and negative direction. i.e) thresholding on the deriviatve is not enough to discriminate between a duck and a jump, although it can discriminate between jump/duck and no command.

3. **Template Matching**: From the graph of the derivative over time, we can see that a jump and a duck have inverted profiles on the graph. If we compute the correlation between a template graph for a jump/duck and the current graph at all timesteps, we would be able to discriminate between a jump and a duck effectively. However, this method cannot be used in real time. As we can see from either graph, a jump/duck occupies a significant portion of time. Computing the correlation between a template feature and the motion of the midpoint face in real time would only result in classification at the end of a jump or duck, which would introduct unwanted latency into the system.

4. **Threshold on Derivative with extra constraint**: The method we implement in our system is a modification of simply thresholding on the derivative of the y position. We threshold on the vertical position, with the added constraint that at any time t, for the sample at that time to be classified as a jump or a duck, the previous N samples of the derivative must be below the threshold. This ensures that only the first period of rapid change in the motion of the midpoint face is captured as a command. N and the threshold are tunable parameters, and in our final system, we set the threshold = 10 pixels/sec and $N = 3$.

The final solution we implement for our feature tracking/classification for the Chrome dino game is as follows:

1. For every frame of video, compute y_t, the midpoint of y-components of all keypoints existing in the frame from the set {left eye, right eye, left ear, right ear, nose}.

2. Compute y'_t, the difference of y_t and y_{t-1}.

3. If $y'_t > 10, y'_{t-1} < 10, y'_{t-2} < 10$ and $y'_{t-3} < 10$: Classify current frame as a duck.

4. If $y'_t < 10, y'_{t-1} < 10, y'_{t-2} < 10$ and $y'_{t-3} < 10$: Classify current frame as a jump.

5. If none of the above conditions are satisfied, classify current frame as no command.

Note that $y' > 10$ corresponds to a duck and $y' < 10$ corresponds to a jump as the image coordinates are zeroed at the top left of the frame.

3.4. Game Interface

The final step before our human-computer interaction system is fully functional is to simulate keypresses to play the two example games selected. In this project we select simple example games that can be run on the Jetson Nano within a browser or through pyGame. However, the system was designed so that during this phase, the Jetson Nano could send data to a game console machine either through serial communication or wirelessly through bluetooth/wifi communication to play more intensive games on a dedicated machine.

In order to simulate keypresses, we utilize a Python package known as pynput, which allows us to press and release arrow keys at the appropriate times in order to simulate keypresses to the example video games.

4. Results

Once the human pose estimator, keypoint classifiers, and game interface are integrated into a single system, we are able to play both of our example games in real time (15+ FPS) through only our gestures in front of the camera. Our systems are robust, performing well across a variety of scenarios, including when keypoints are missing and erroneous pose estimation. The games are responsive and fully playable from only gestural input, demonstrating little to no loss of playability when moving from keyboard and mouse input to the gestural command input through our human-computer interface device.

5. Conclusion and Future Works

The goal of this project was to develop a system that can map human gestures through a video stream into usable computer command input, replacing how we currently interact with computers through keyboards, mice, and joysticks. Furthermore, we aimed to develop a system that places little to no computational expense on the computer

playing the game. Our system demonstrates that we accomplish this goal in real time, by playing two example video games using gestural inputs, with computation and realtime inference done on an edge device, the NVIDIA Jetson Nano.

In developing and testing this system, we note a few major areas in which the system can be improved in the future. We notice that pose estimations using PoseNet on the Jetson nano fails to detect keypoints around the wrist dependent on the position and condition of the user's hands. We notice that detection is more stable over time when the user's hands are open rather than in the closed fist position, although keypoints are still correctly identified in both positions. One area of future work that can address this problem is to retrain pose estimation models using a custom dataset with more examples of situations such as mimicing steering motions.

Other areas of future work that can improve our system and bring it closer to the goal of a general purpose human computer interface that can mimic a wider range of commands include more general keypoint classifiers using machine learning techniques that can be easily expanded to a wider range of commands. In addition, a communication link between the Jetson Nano and a gaming device can be implemented, allowing the user to play games on other devices, using our human-computer interface only as an intermediate step.

References

[1] Pose estimation. `https://www.tensorflow.org/lite/examples/pose_estimation/overview`, 2022.

[2] Z. Cao, G. Hidalgo Martinez, T. Simon, S. Wei, and Y. A. Sheikh. Openpose: Realtime multi-person 2d pose estimation using part affinity fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.

[3] Dustin Franklin. jetson-inference. `https://github.com/dusty-nv/jetson-inference#readme`.

[4] Vikas Gupta. Deep learning based human pose estimation using opencv. https://learnopencv.com/deep-learning-based-human-pose-estimation-using-opencv-cpp-python/, 2018.

[5] Charlie Liu. Gesture detection with a raspberry pi. 2017.

[6] George Papandreou, Tyler Zhu, Liang-Chieh Chen, Spyros Gidaris, Jonathan Tompson, and Kevin Murphy. Personlab: Person pose estimation and instance segmentation with a bottom-up, part-based, geometric embedding model, 2018.

[7] Adrian Rosebrock. Simple object tracking with opencv. 2018.