CS 283 Final Project
Pipeline for Improving Accuracy of Hand Tracking with a Poorly Trained Haar Cascade Detector

Kenny Yu, HUID: 30798260

Fall 2012

1 Introduction

What if you had the power of Microsoft's Kinect in your webcam? What if you could control applications simply with a swipe of your hand? Detecting and tracking hands typically requires a good detector, which requires large amounts of good data (usually at least 5000 images !!!CITE!!!) and a tremendous amount of computation time to train the classifier. In this paper, we trained our hand detector with only 250 images, generated from 10 positives and 100 negatives. Despite this poorly trained Haar cascade classifier, we provide a pipeline to improve accuracy during hand tracking, achieving comparable results to tracking with a well-trained hand detector. Using this pipeline, we create a sample application !!!CITE!!! in which the user can control Google Maps !!!CITE!!! by simply moving one's hand. See our source code in !!!CITE!!! for our pipeline and server to run this application.

2 Methods

2.1 Training the Hand Detector

To generate our hand detector, we took 10 pictures of our right hand using a webcam from a Macbook Pro 2011 model, and we cropped the images to leave only our hand in the images. Our pictures consisted of our right hand with the palm facing forward and all fingers together. See Figure 1 for the the images we used as positives. We used the first 100 images from !!!!CITE!!!!! as negatives. We used OpenCV's `opencv_createsamples` !!! CITE !!!!! utility to generate 250 90 $\times$ 50 pixel samples from this set of 10 positives and 100 negatives, resulting in images where our positive images have been warped by homographies and placed randomly into our negative images. We used OpenCV's `opencv_haartraining` tool to train our Haar cascade classifier, setting our min hit rate to 0.999, our max false alarm rate to 0.5, and running 14 stages in our cascade. This took about a day to complete, and the resulting cascade is located in `cascades/hand_front.xml`.

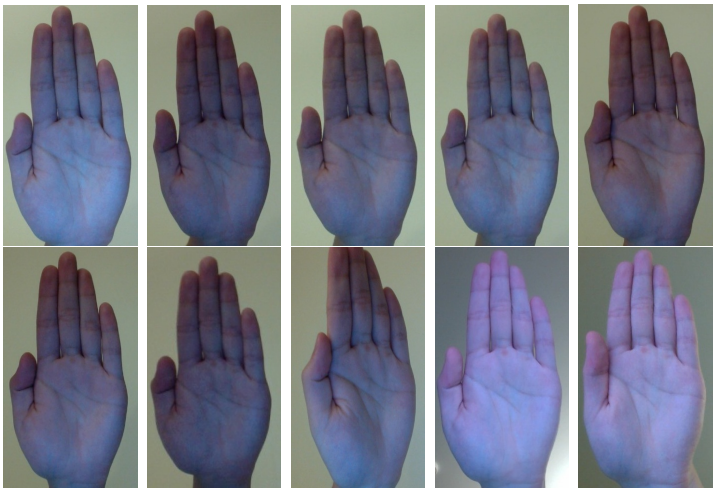


Figure 1: The 10 positive images used to train our hand detector. We generated 250 samples from these, each of size 90 pixels high and 50 pixels wide.

2.2 Building the Pipeline

See `detect.py` for the different components of our pipeline, and see `pipeline.py` for the various pipelines. For our pipeline, we used a frame size of 320 pixels wide and 240 pixels tall. We ran our detector without any preprocessing, and noted that we had many noisy detections, as shown in Figure 2(a). The yellow boxes indicate multiple possible detected hands, and the green one represents the largest one in the overall frame. We noted that our detector often mistook faces as hands, probably because of similar color tone. Using this heuristic, we decided to remove faces, and this generally gave us better results, as shown in 2(b). We used OpenCV's frontal face detector in `cascades/haarcasacade_frontalface_alt.xml`, which works very well in practice.

Next, we noted that our detection was sensitive to noisy non-solid backgrounds. To account for this, we implemented background subtraction. To do this, we kept a history of the past n frames, and we took the difference of the intensities of each pixel in the current frame and the oldest of the n frames, and thresholded the difference. We found that setting our threshold to 20 worked best, and we found that the number of frames $n = 20$ worked quite well. See Figure 2(c) to see an example of the result of applying background subtraction. Background subtraction, however, had several issues. The dark patches left in the middle of the hand would sometimes prevent valid detections, and the randomly scattered dark patches also led to some false positive detections.

To fix the latter issue, we used a simplified version of a Kalman Filter !!!! CITE !!!! to predict a bounding box window of where the hand would likely be in the next frame. We made the assumption that a hand typically moves in a smooth manner, and so using the velocity and current position of the hand will give us a pretty good prediction for where the hand will be in the next frame.

We defined our state $\mathbf{x_k}$ at step k to be a 4-vector:

$$\mathbf{x_k} = (x, y, v_x, v_y)^\top,$$

where x, y is the current position (center) of the hand, and v_x, v_y are the x and y components of the velocity of the hand. From this definition of our state, we naturally defined the next state (prediction) $\mathbf{x_{k+1}}$ to be:

$$\mathbf{x_{k+1}} = (x + v_x, y + v_y, v_x, v_y)^\top,$$

where we obtained the new x, y components by adding the corresponding components in the velocity. Thus, our transition matrix $\mathbf{A}$ satisfies the equation

$$\mathbf{x_{k+1}} = \mathbf{A}\mathbf{x_k},$$

where

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

When we first attempted to use this Kalman filter, we had used error, process noise, and measurement noise covariance matrices, however, we discovered that reducing the Kalman filter to a simple Markov process improved runtime performance and greatly simplified the overall model.

In our pipeline, we used our Kalman filter to predict a state $\mathbf{x_{k+1}} = (x, y, v_x, v_y)$, where (x, y) represents our prediction of the center of a $h \times w$ bounding box where we would probably detect our hand. In practice, we discovered that a $h = 180, w = 100$ worked best. Given this window, we applied our hand detector only within this window. See Figure 2(d) for the result of this hand detection, and see the blue box in Figure 2(e) to see the Kalman prediction window relative to the original frame.

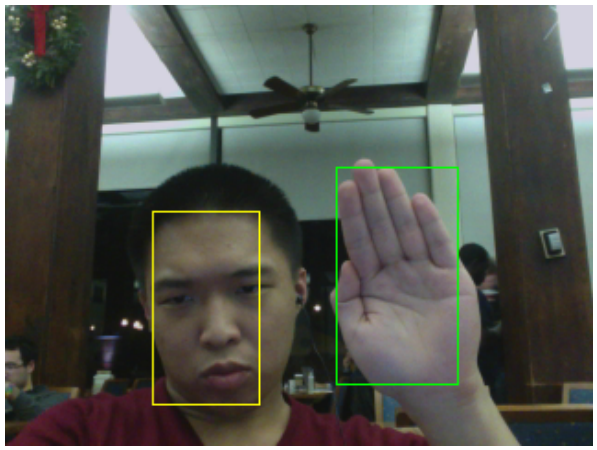
We detected the corners within the detected hand, and then we used Lucas-Kanade !!! CITE !!! to compute the optical flow of these corners. We summed the overall displacements to obtain the overall velocity $(\hat{v_x}, \hat{v_y})$. To correct our estimate, we set our corrected state $\hat{\mathbf{x_k}}$ to be:

$$\hat{\mathbf{x_k}} = (\hat{x}, \hat{y}, c\hat{v_x}, c\hat{v_y})^\top,$$

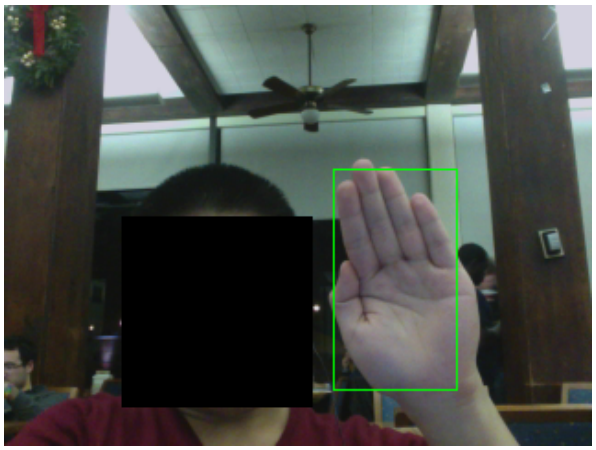
where $(\hat{x}, \hat{y})$ is the center of the detected hand, and c is constant we chose to scale back the overall direction. If c is too large, then our window will overshoot the position of the hand. If c is too small, then our hand will move past the prediction window and will not be detected. In practice, we discovered that $c = 0.02$ works best. We then obtained our prediction $\mathbf{x_{k+1}}$ for the next frame by using our corrected state $\hat{\mathbf{x_k}}$:

$$\mathbf{x_{k+1}} = \mathbf{A}\hat{\mathbf{x_k}}.$$

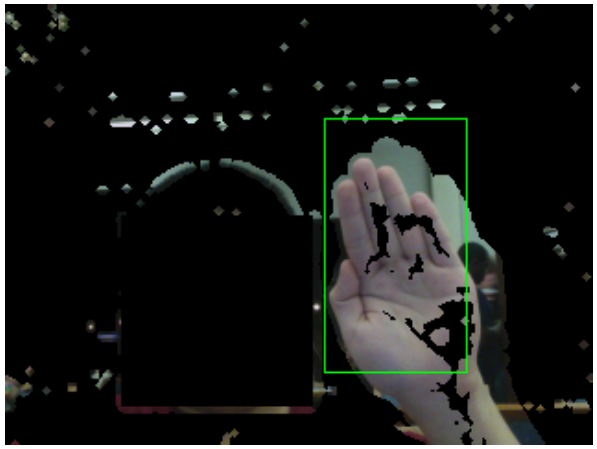
When there are no detected hands, we set our Kalman prediction to be the entire original frame. Our final pipeline is listed in Algorithm 1.



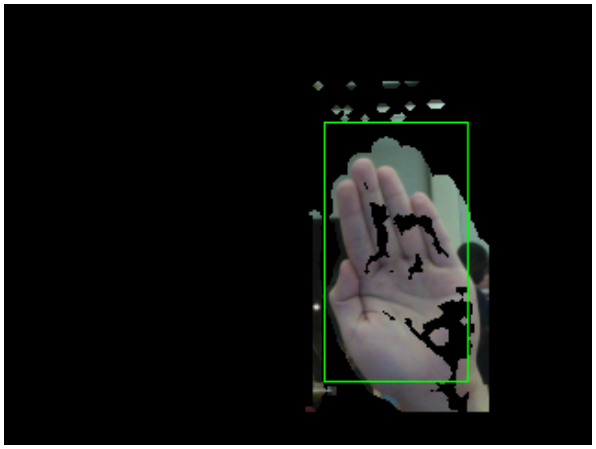
(a) Our hand classifier had many noisy detections.



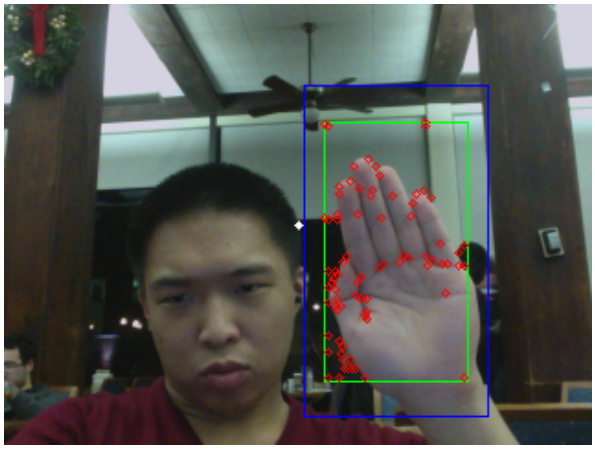
(b) Face Removal



(c) Background Subtraction



(d) Kalman Prediction Window



(e) Final Detection and Optical Flow

Figure 2: Our pipeline and hand detections at each stage. Green box - largest detected hand. Yellow boxes - other detections. Blue box - Kalman filter prediction. Red dots - corners.

Algorithm 1 Pipeline for Improving Hand Tracking Accuracy with a Poorly Trained Hand Detector.

Given the current frame and the next frame:

1. **Face Removal.** Apply OpenCV's frontal face detector and remove all detected faces.
2. **Background Subtraction.** Eliminate the background through background subtraction.
3. **Kalman Prediction.** Use our simplified Kalman filter to predict the likely bounding box of the hand in the current frame. If the previous frame had no detected hand, then our Kalman filter will estimate the entire frame as the bounding box.
4. **Hand Detection.** Apply our hand detector within the bounding box (blue box) predicted by our Kalman Filter. If there are multiple detections (yellow boxes), take the first hand with the largest bounding box (green box)
5. **Corner Detection.** If there are no detections, then output $(0, 0)$ as our velocity. If there is a detection, compute the corners (red points) within the bounding box of our hand.
6. **Optical Flow.** Use Lucas-Kanade to compute the optical flow of the detected corners, finding the corresponding points (if they exist) in the subsequent frame. Sum all the vector displacements (green vectors) to compute the overall velocity of the hand (white vector).
7. **Kalman Update.** Update our Kalman Filter with the measured position and velocity of our detected hand, and repeat on the next frame.

2.3 Building the Google Maps Application

We used our pipeline to create a simple application in which the user can control Google Maps by moving one's hand. We follow the model proposed in !!!CITE!!! to create our application. We used Chrome's WebRTC ability to capture a video stream from a connected webcam. Using web sockets !!!CITE!!!, we send a constant stream of frames to our server that is running our pipeline. We used Tornado 2.4.1 !!!CITE!!! to run our server, and we used OpenCV 2.4.1 !!!CITE!!! to implement our pipeline. The server runs the pipeline and outputs the annotated frame and the overall computed direction of the hand, and the client then uses this information to update the map accordingly. A live running instance of the application can be publicly accessed at `http://goo.gl/pcNxG`. See Figure 3 for a screenshot of the application in action. The green vectors represents the direction of the corners, and the white vector is the sum of the green vectors, representing the overall direction of the hand.

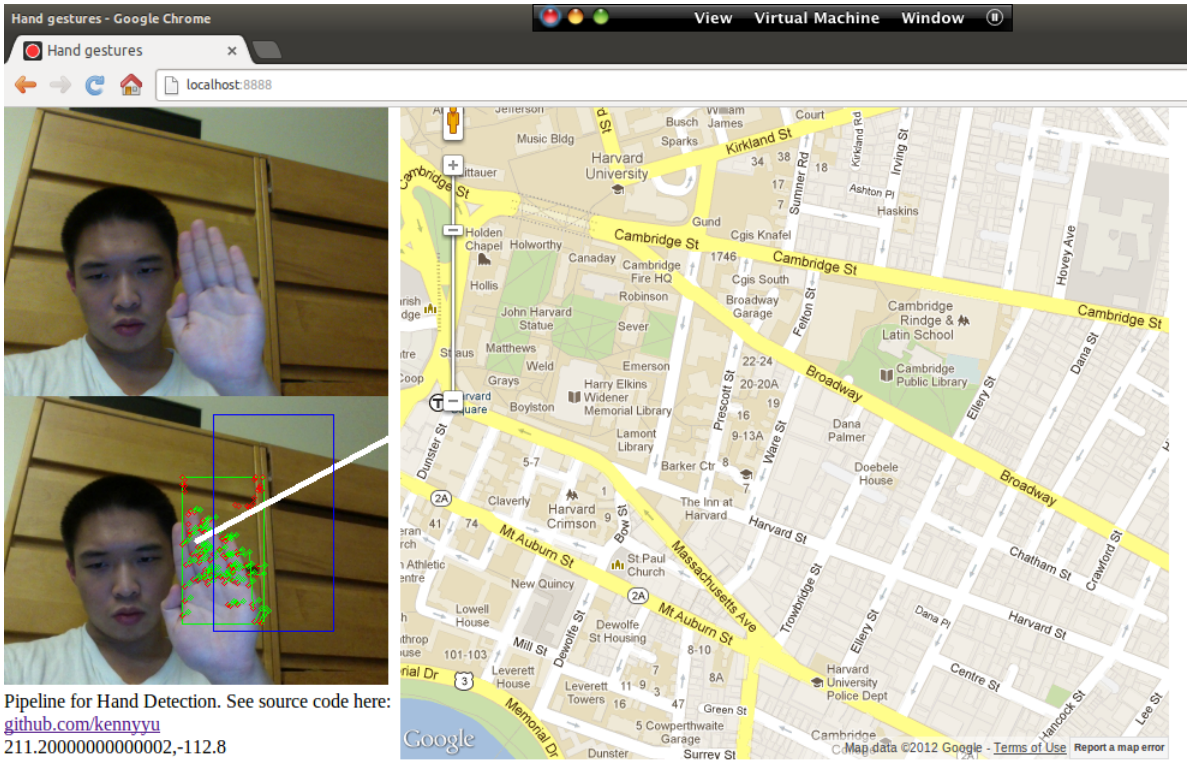


Figure 3: Screenshot of our Google Maps Application in action. The green vectors represents the direction of the corners, and the white vector is the sum of the green vectors, representing the overall direction of the hand.

3 Results

Background subtraction no good, too slow, better not to use it.

4 Conclusions

5 References