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

Kenny Yu (kennyyu@college.harvard.edu)

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 [2]) 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 [9], 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 B in which the user can control Google Maps [4] by simply moving one's hand. See our source code in A or 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 [8] as negatives. We used OpenCV's `opencv_createsamples` [2]utility to generate 250 90×50 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`.

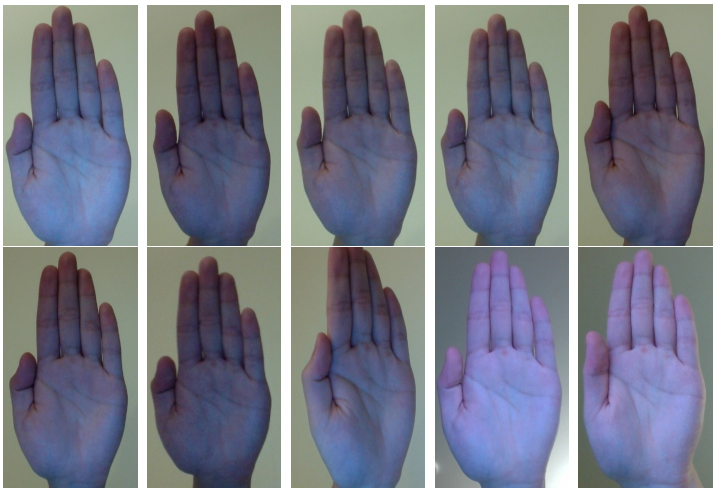


Figure 1: The 10 positive images used to train our hand detector. We generated 250 samples from these, each of size 90 pixels high and 50 pixels wide.

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 [12] 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.

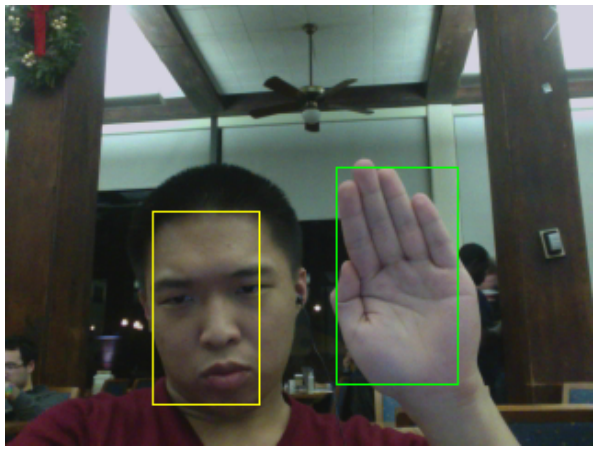
We detected the corners within the detected hand, and then we used Lucas-Kanade [5] to compute the optical flow of these corners. We summed the overall displacements to obtain the overall velocity $(\hat{v_x}, \hat{v_y})$. To correct our estimate, we set our corrected state $\mathbf{\hat{x_k}}$ to be:

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

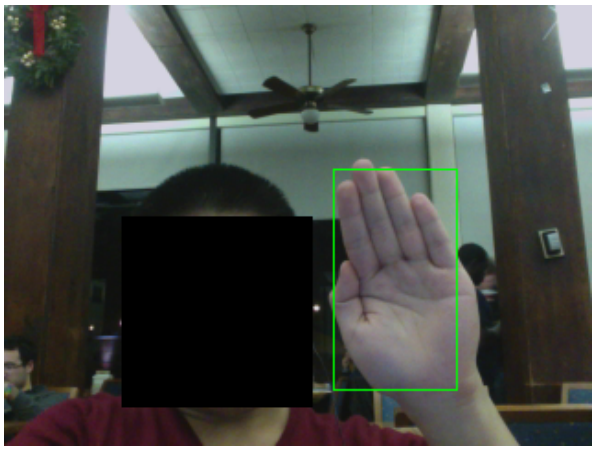
where $(\hat{x}, \hat{y})$ is the center of the detected hand, and c is constant we chose to scale back the overall direction. If c is too large, then our window will overshoot the position of the hand. If c is too small, then our hand will move past the prediction window and will not be detected. In 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 $\mathbf{\hat{x_k}}$:

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

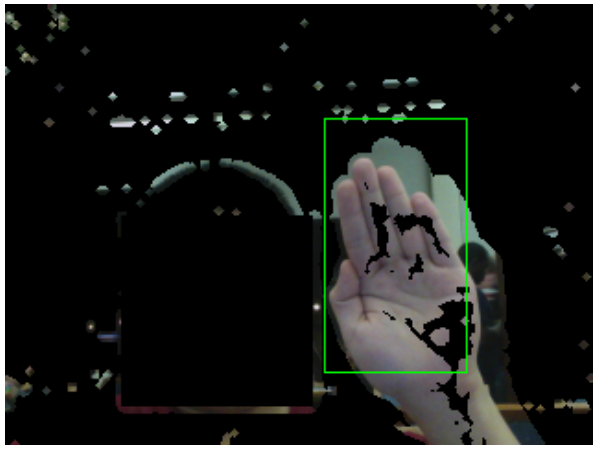
When there are no detected hands, we set our Kalman prediction to be the entire original frame. Our final pipeline is listed in Algorithm 1.



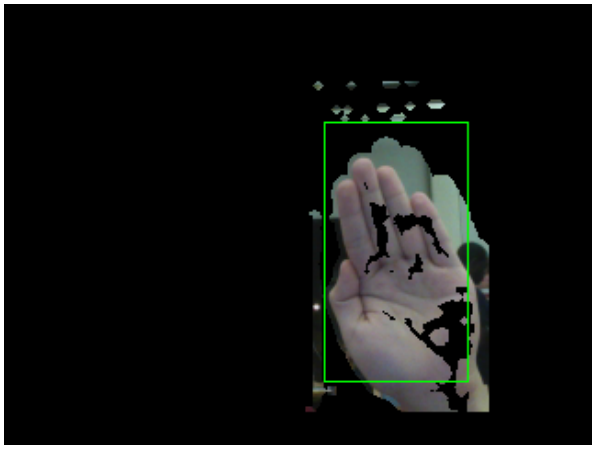
(a) Our hand classifier had many noisy detections.



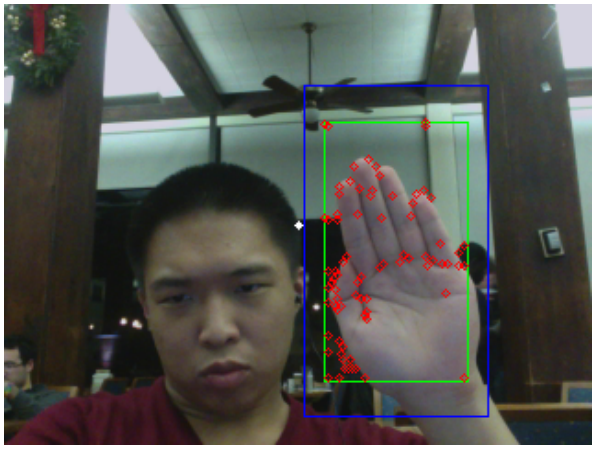
(b) Face Removal



(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 Accuracy of Hand Tracking 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 by subtracting the intensities in the current frame and previous n-th frame, and threshold the difference.
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 [3] to create our application. We used Chrome's WebRTC [10] ability to capture a video stream from a connected webcam. Using web sockets [11], we send a constant stream of frames to our server that is running our pipeline. We used Tornado 2.4.1 [7] to run our server, and we used OpenCV 2.4.1[6] 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`. We set up this server on a medium Amazon EC2 instance. 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.

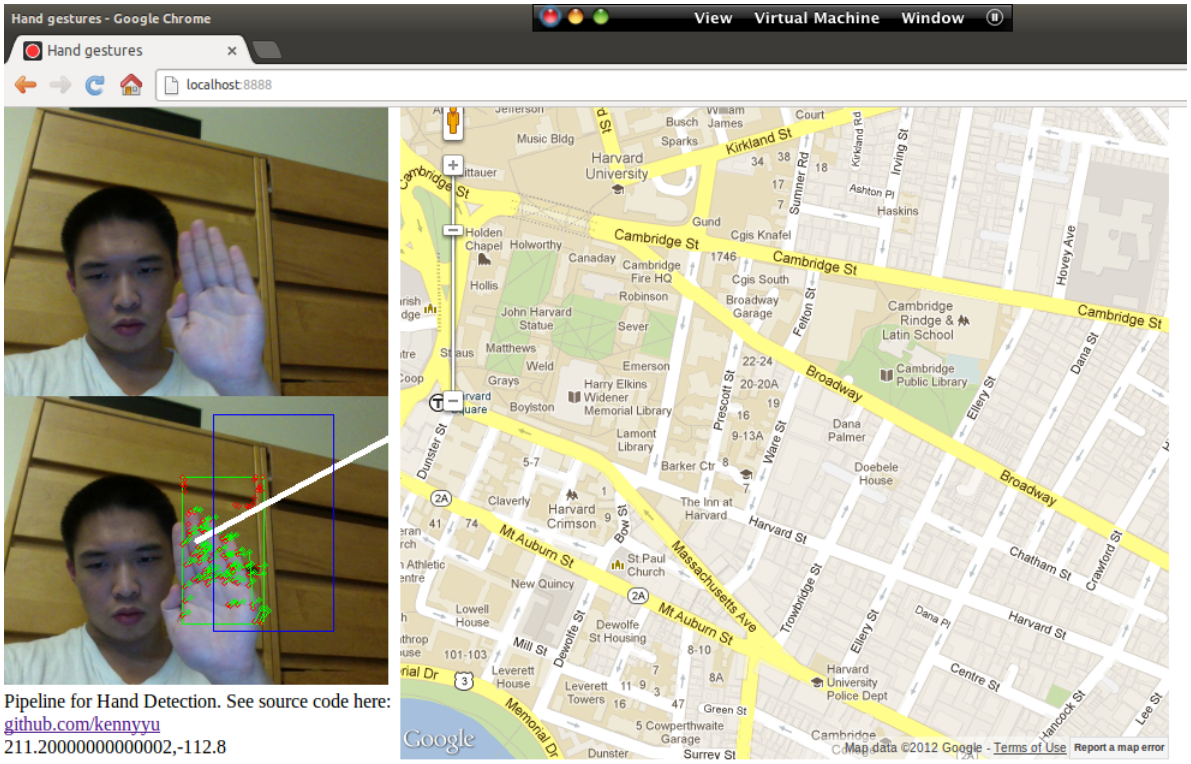


Figure 3: Screenshot of our Google Maps Application in action. The green vectors represents the direction of the corners, and the white vector is the sum of the green vectors, representing the overall direction of the hand.

3 Results

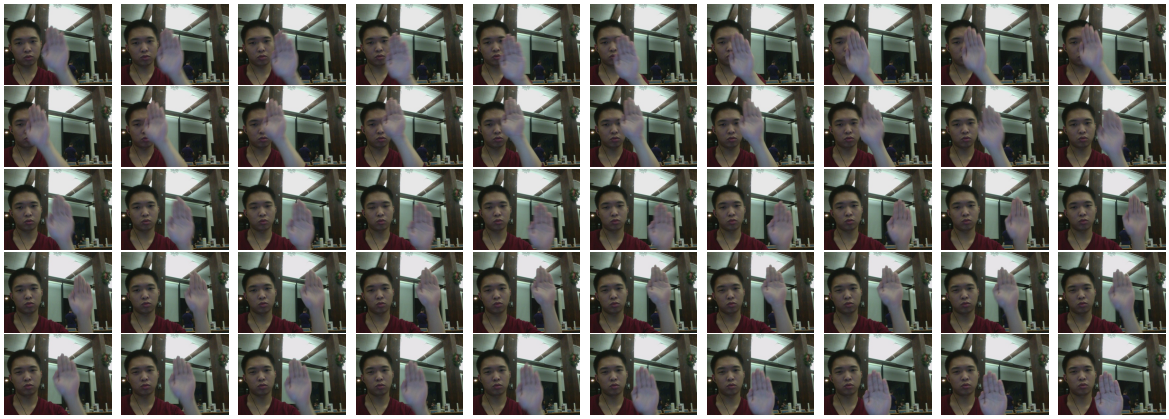


Figure 4: The 50 frame sequence we used to test the components of our pipeline.

To gauge the various components of our pipeline, we created a 50-frame sequence, shown

in Figure 4. We created 8 pipelines, generated by our choices of whether to remove faces, apply background subtraction, and use a Kalman filter. We always perform optical flow to measure the direction of the hand. After generating the images, we went through the images by hand to count the number of true and false positives. We define a true positive when we have a green box around the hand, and we define a false positive whenever we see a yellow or a green box surrounding something that is not a hand. See Table 5 for our results, and see the `data` subdirectories for the annotated images for each pipeline.

Pipeline Type	FPS	True Pos.	TPR	False Pos.	FPR	Average v_x	Average v_y	Average Width	Average Height
None	25.348	46	0.939	35	0.714	-18.772	20.701	73.061	131.389
Face	13.373	33	0.673	0	0	-17.565	28.768	49.429	88.837
Kalman	22.841	45	0.918	4	0.0816	-31.605	30.921	61.735	110.878
BG	21.478	37	0.755	32	0.653	-16.85	-7.266	76.551	137.571
Face, Kalman	11.795	32	0.653	0	0	-28.157	46.57	42.429	76.163
Face, BG	12.423	31	0.633	2	0.0408	-15.287	16.235	51.49	92.571
BG, Kalman	19.841	23	0.469	25	0.510	-0.476	-5.81	66.265	118.89
All	10.888	28	0.571	0	0	-11.663	24.01	46.122	82.939

Figure 5: Results of running all the combinations of components in our pipeline on the same 50 frame sequence.

Applying our hand detector alone had a very high true positive rate, but also generated many false positives as one would guess. We note that performing face removal using OpenCV's frontal face Haar cascade significantly decreases the frame rate in all cases (and hence, slower performance), but dramatically decreases the number of false positives.

We note that a background subtractor generates many false positives when applied alone and with a Kalman filter. Furthermore, we noted that using background subtraction does not detect a hand if the hand stays perfectly still–overtime, the hand will blend into the background and will not be seen by our detector.

A Kalman filter generates the fewest false positives for the number of true positives it detected, and when combined with face removal, generates no false positives. Combining all three of these components generates no false positives, but detects fewer true positives than only applying face removal and using a Kalman filter.

Hence, if our main priority is to eliminate false positives, then we only need to apply face removal and use a Kalman filter. If our main priority is to generate as many true positives while maintaining a fairly low false positive rate, then we should only use Kalman filter. For our maps application running on a medium sized EC2 instance [1], we discovered that running the entire pipeline was too slow to be usable, and that running only the face removal and Kalman filter components made the application much faster and a lot more responsive to user movement.

We note the wide discrepancies in average velocity and average hand size. Thus, the components we use in our pipeline dramatically affects the resulting output in the maps application.

4 Conclusion

In this paper, we provided a method to quickly train a hand detector using only a few images, and achieved comparable hand tracking accuracy to using a perfect hand detector. We also described and built a system for running our pipeline on a real time application in which the user controls Google Maps by moving one's hand. For future work, we would like to explore how to generalize this process.

We would like to explore how well this pipeline works for different kinds of hand positions, e.g. fists, detecting certain digit configurations, and more. Furthermore, we would like to explore other components that can be added to our pipeline to further decrease the number of false positives and increase the number of true positives, while maintaining a relatively high frame rate to actually be usable in applications. A limitation of this current pipeline, however, is that only one hand can be tracked. It becomes unclear how to generalize some of the components of our current pipeline when we attempt to track multiple hands in the frame, e.g. it is unclear how many Kalman prediction windows to use and how we use these multiple windows.

Thus, by using less training data and a bit more computation power, we hope that commodity webcams will one day be able to emulate many of the features offered in a Kinect.

5 Appendix

A Source Code

To checkout the source vode, visit `https://github.com/kennyyu/cs283-project`. See the `README` for information on how to run the various pipelines, the server, and analyzer scripts. The generated cascades are located in the `cascades` directory, and the images we used to train our detector are located in the `data/positives` directory.

B Maps Application

To view the live running maps application, visit `http://goo.gl/pcNxG`. You must use Chrome, have enabled web sockets (see `chrome://flags`), and have a connected webcam. The application is currently running on a medium Amazon EC2 instance [1].

References

[1] Amazon Elastic Compute Cloud. <http://aws.amazon.com/ec2/>.

[2] "Cascade Classifier Training". OpenCV 2.4.9 Documentation. Accessed 12 Dec. 2012. <http://docs.opencv.org/trunk/doc/user_guide/ug_traincascade.html>.

[3] Dirksen, Joe. "Face detection using HTML5, javascript, webrtc, websockets, Jetty, and OpenCV". Smartjava.org. 19 Apr. 2012. <http://www.smartjava.org/content/face-detection-using-html5-javascript-webrtc-websockets-jetty-and-javacvopencv>.

[4] Google Maps. Google, Inc. <http://maps.google.com>.

[5] Lucas, Bruce D. and Takeo Kanade. "An Iterative Image Registration Technique with an Application to Stereo Vision". Proceedings of Imaging Understanding Workshop, 1981, pp. 121-1380.

[6] OpenCV 2.4.1. <http://opencv.org/>.

[7] Tornado Web Server 2.4.1. <http://www.tornadoweb.org/>.

[8] Seo, Naotoshi. "Tutorial: OpenCV haartraining". <http://note.sonots.com/SciSoftware/haartraining.htm>.

[9] Viola, Paul and Michael Jones. "Rapid Object Detection using a Boosted Cascade of Simple Features". Conference on Computer Vision and Pattern Recognition (CVPR), 2001, pp. 511-518.

[10] "Web Real-Time Communications Working Group Charter". W3C. 4 Dec, 2012. <http://www.w3.org/2011/04/webrtc-charter.html>.

[11] "The WebSocket API". W3C. 4 Dec, 2012. <http://dev.w3.org/html5/websockets/>.

[12] Welch, Greg and Gray Bishop. *An Introduction to the Kalman Filter*. University of North Carolina at Chapel Hill. SigGraph 2001. Ch. 4. pp. 19-29. <http://www.cs.unc.edu/~tracker/media/pdf/SIGGRAPH2001_CoursePack_08.pdf>.