Human Detection and Localization from an Unmanned Aerial Platform

Joshua Cutolo, Nicholas Kennedy, Joshua Pinos

April 30, 2015

Abstract

Detection, tracking, and localization on human subjects from an MAV equipped with a monocular camera is only recently achievable using recent advances in multiple disciplines of computer science. This paper and accompanying software begin to form a viable solution to this problem. By using Histograms of Gradients (HOG)[Dalal and Triggs, 2005], we can calculate feature vectors that allow us to classify humans in a camera frame. Large-Scale Direct Monocular SLAM (LSD-SLAM)[Engel et al., 2014] provides a robust solution to camera localization without external input. We can then project a point located within the feature vector detection window into world coordinates. Using these world coordinates, we can improve tracking of the human subjects compared to overlapping windows.

1 Introduction

An unmanned aerial platform provides a unique and complex encumbrance to object recognition and localization of said object, with the vacillation of the drone

2 Feature Detection

2.1 Data Collection

In order to begin the process of feature detection, we first had to gather videos, to collect our training data, using the provided drones. We decided to use Ubuntu along with ROS in order to capture the data efficiently using our computers and a USB device. After a week of learning how to use the drones, we figured out that using our phones was not allowing us to control the drone optimally while in flight. After some research, we learned that ROS, installed in Ubuntu, would allow for us to record a more optimal video that was suited for needs. After configuring the settings of ROS, as well as the virtual machines running Ubuntu, we managed to connect wirelessly to the drones and were able to view the video feed on our computers. In total, we gathered a couple of videos that gave us a sampling size of about 3784 total images, 1849 positive and 1935 negative images. Positive images are images in a video that contain a human while the negative images are the ones that do not. However, in order

to try and better train our SVM, we decided to use approximately 800 negative images that contained other objects including trees, buildings, and light posts. In our opinion, we believed that this would help the SVM in detecting humans more efficiently due to the fact the SVM would view the other objects in the images as "negative" and therefore not humans. Our goal was to filter out as many different objects that were not humans as we could. An SVM, also known as a Support Vector Machine, is a model that classifies training data.

2.2 Algorithm Design

After collecting our data, we began designing the algorithms that would assist us in training and testing the SVM models, and that would implement the HOG detector as well. A HOG, or Histogram of Oriented Gradients, are feature descriptors used in image processing and computer vision in order to detect objects in an image. It splits an image into smaller and smaller connected regions, and compiles a histogram of gradient directions for each pixel in an image which, in combination with one another, creates the feature descriptors. From these descriptors, a feature vector is created. The HOG algorithm has an effective track record in detecting humans with adequate accuracy so we chose to implement this method as our main method of detecting human being in a video.

To begin with, the first program we had to design was an image cropper program that allowed the user to select different images in a video and separate them as either positive or negative images. For this program, we decided to use an image size of 64x128 because it was an optimal image size for working with OpenCV, the computer vision software we used during the project. The program is a simple python script that allows the user to cycle through the video frames from one of their videos in a specified directory, and select a 64x128 region of interest and designate that area as either a positive or negative image and saves them in a file on the user's computer with a $p.jpeg$ extension if the image is positive or a $n.jpeg$ extension if the image is negative. Also, the video cropper displays the frame number the image was found at in the file extension as well.

Next, we had to create the feature vectors from the different images we obtained during the image cropping portion of the project and place them in a CSV file, in order to begin training the SVM. In order to create the feature vectors, we used the process of HOG as explained previously. The program we created took each image one by one from the file and computed a feature vector of that image using the $hog.compute()$ method and then placed it into the newly created CSV file. From there, it was finally time to begin training our SVM.

Using the csv file created during the feature vector process and also imputing any feature vectors in the file that were equal to zero with the NaN object, we split the feature vectors into both training and testing datasets. By using the $train\_test\_split()$ function imported from sci-kit learn, we were able to randomly split up our data, with 30 percent of it for testing and 70 percent of it for training. At this point the program proceeds into two concurrent steps: Training multiple models using k-fold cross validation and training of a general model, which will be implemented later on in the real-time detection algorithm. For the first path, the training data will be manipulated for k-fold cross validation with 5 folds. Five SVM models are instantiated, trained, and validated with their respected five subsets, created by the cross validation function. As these SVM models

are trained and validated, a general SVM model is fitted with the training dataset and tested with the testing dataset. The accuracy of each SVM after the validation is collected and the mean is reported back. As for the general SVM, a classification report and confusion matrix is generated reported back as well. The results of these experiments are discussed in detailat the conclusion of this discourse. As a precautionary measure and as a comparison baseline, an SVM model from OpenCV's library is created, trained, and tested with the same dataset implemented by the general SVM. Both the general SVM model and OpenCV's SVM model are stored as a python pickle file and xml document, respectively, for later use in the real-time detection algorithm.

After the training of the SVM models were complete, the models are then tested on video files for their accuracy. The two models from the previous steps are loaded up and converted to a datatype that will be passed to the $detectMultiSCale()$ method, along with the default HOG classifier that is built in with OpenCV. This method scans a given image with a moving box, calculating the feature vectors from the region of interest defined by the moving box, and determines if a human was detected, using the model that was given as a parameter. If a human is detected, the region of interest's bounding box is returned and displayed on the video, for visual analysis of the algorithm. We tested all classifiers on the videos that the samples were extracted from.

3 Camera Localization

LSD Slam provides a robust, real-time, method for camera localization which is essential to accurately projecting a point from camera coordinates $\mathbb{R}^2$ to world coordinates $\mathbb{R}^3$.

LSD-SLAM improves on feature based monocular SLAM. Feature based SLAM uses two steps. First, features are detected and matched with a previous frame. Second, a transform between the features is calculated which results a perspective projection matrix P, and a matrix of homogeneous coordinates $H\mathbb{R}^2$ of the features.

LSD-SLAM circumvents this 2-step method by using the image intensities. This allows the entire image to be used, thus increasing fidelity of the resulting map. Engal describes a drift error of 0.50 cm/s for the translation vector, and 0.31/degrees a second for the rotation matrix. This compares with 8.2 cm/s and 3.2 deg/s, respectively, for the PTAM method. It is notable that LSD-SLAM is more accurate than position data gained from double integrating accelerometer data, where an angle error of just 0.1 degrees results in a position error of 170 cm/sec. Most low cost GPS solutions, while not subject to drift, only provide position data $\pm$ 3m along the surface of the earth and $\pm$ 10m for elevation data.

We are unable to use LSD-SLAM directly to calculate world coordinates for moving objects as LSD-SLAM is limited to localization of rigid structures. Modification of the LSD-SLAM algorithm to localize non-rigid was not possible with the knowledge of the team and time constraints for the project. It did however provide an accurate queries describing the rotation vectors, as well as translation vector.

LSD-SLAM works best when the camera is translated across a scene, and uses a camera with a high frame rate. Our data set was a video of a scene where the camera is mostly stationary, but with acceleration along its rotation vectors.

Our camera operated at only 30fps. Engal recommends a camera with a frame rate of 60fps or higher. This caused issues where tracking was lost. We default to detecting same features by overlapping windows in camera coordinates in this situation. However, LSD-SLAM is robust enough, that when the camera has a scene that was previously viewed in frame it is able to reacquire tracking. This re-acquisition did come with a non-significant translation error, but was still within the error a conventional GPS unit would provide.

4 Feature Localization

We separated the problem into three parts, camera localization, feature detection, and feature localization and tracking. To better discretize the experiment both the data from the feature detection and the camera localization were prepared ahead of time and stored in a human readable format.

The first step in implementation was reading in this data. We took much care to design the localization and tracking code so that it can easily be modified to run in a real-time scenario. That is; the program has no knowledge of the data included in the frames past the current frame that is being processed.

4.1 Projection to $\mathbb{R}^3$

Projection of camera coordinates $C = [u, v]$ to world coordinates $W = [x_w, y_w, z_w]$ is possible knowing the camera's quaternion $R_q = [x_r, y_r, z_r, w_r]$ which describes its rotation in space, translation vector $t = [x_w, y_w, z_w]$ as well as the intrinsic parameters of the camera K.

The conversion from R_q to R, the rotation matrix is given by:

$$R = \begin{bmatrix} 1 - 2q_y^2 - 2q_z^2 & 2q_xq_y - 2q_zq_w & 2q_xq_z + 2q_yq_w \\ 2q_x * q_y + 2q_zq_w & 1 - 2q_x^2 - 2q_z^2 & 2q_yq_z - 2q_xq_w \\ 2q_xq_z - 2q_yq_w & 2q_yq_z + 2q_xq_w & 1 - 2q_x^2 - 2q_y^2 \end{bmatrix}$$

The Camera's Intrinsic Matrix, K, is given by:

$$K = \begin{bmatrix} \alpha_x & \gamma & u_0 \\ 0 & \alpha_y & v_0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$P = K(R^T - R^T t)$$

We multiply the 2nd column of P by z_w, then subtract the 3rd column of P from that result

*z_w is the estimate of the height of the feature above the world plane where $z_w = 0$

$$M_1 = z_w(P_{*2}) - P_{*3}$$

$$M_2 = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} M_1^{-1} t^T$$

$$x_w = \frac{M_2[1]}{M_2[3]}, y_w = \frac{M_2[2]}{M_2[3]}, z_w = z_w$$

4.2 Pruning

The feature detector included a significant amount of noise. There are many generated windows that have no possibility of containing a human. We start our pruning by eliminating detection windows that are overlapping. We further prune the windows that have an area greater than one standard deviation from the windows in our data set. During this pruning, an attribute set represents the detection window's world coordinates. This step increases the speed at which we can find the nearest window.

4.3 Overlapping Windows Method

To track moving features absent of any information of the feature in world coordinates we are forced to rely on the position of the widow in camera coordinates. This method is unsuitable in situations where the camera's position is unstable. The AR-Drone's camera has a 92-degree field of view. When the image is downsampled to a resolution of 640x352, this FOV translates to nearly 7 pixels for every degree of movement. A movement of just 20 degrees will result in the image shifting 140 pixels. Tracking features based upon only the camera coordinates introduces a large amount of error and provided poor results.

4.4 Nearest Projected Point Method

By first projecting the feature locations to world coordinates we can negate the effects of the camera movement. This method begins by looking back to the previous frame. We iterate through the coordinate locations for every feature. If we find a point, that is within a threshold we reflect that feature's attributes: ID, and number of times detected onto the new box. We then mark that box so that it may not be used again. If no suitable match cannot be found in the previous frame, we continue this method until a suitable match is found, or we reach a pre-determined recursion threshold. At this point, we mark the feature as newly detected and assign it an unused ID number. We can achieve a time efficiency of $O(n\ logn)$. This method fails when there are two features very near each other, or two features cross paths.

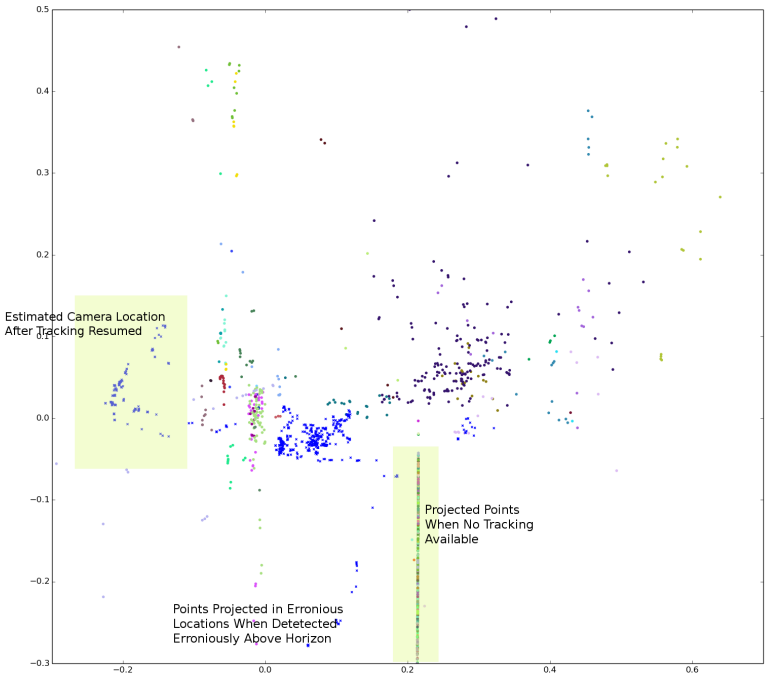
There is a significant amount of error introduced by relying on the coordinates of the detection window. The windows are very unstable, their size changes from frame to frame, or may disappear completely for several frames.

A change of just a few pixels of the boxes dimension or location along the image y-axis can translate into a very large change in the world coordinates that are calculated for the feature. As a point in camera, coordinates moves from the bottom of the camera frame to the top the world coordinates change by the square of the camera coordinates along the cameras y-axis.

The windows also contain no information about the elevation of the feature. The lack of elevation data causes us to make a false assumption that all of the features lie on a flat plane. This assumption causes the features that are lower in elevation projection to be closer to the camera's location while features that are higher in elevation appear farther away.

4.5 Final Pruning

Figure 1: Projected Coordinates $\in [x, y]$



After the matching is complete, the resulting set still includes far too much noise. We further refine this set by only displaying the features that have been detected for constant number of frames. Through experimentation, it was clear that 15 detections produced results that achieved the best balance between throwing away unneeded information and fidelity of the result. We also maintained a library of "hot" boxes, which was a set of boxes that have been active for a constant number of frames. These boxes, absent of any new boxes with a matching identification, were redrawn on the screen using the previous camera

coordinates. The "hot-box" method solved the issue where detection windows would disappear from frame to frame. It was attempted to redraw the boxes by projecting their world coordinated back to camera coordinates, but there are some unresolved issues with the algorithm that handles this calculation. Results of the projected coordinates are seen in *fig 1.*

5 Future Work

5.1 Feature Detection

A solution to HOG detection in the future might be using the language of C++ along with OpenCV in order to use the full power of a GPU on a computer. This would allow for much faster video speed and possibly even better HOG detection. After testing the C++ functions for HOG detection on an older computer, we were able to detect humans through a webcam in real time. Also, another solution might be to use Haar Cascades and Local Binary Patterns like we talked about in class; However, OpenCV came out with a newer version on the 24th of April. We are unsure at this point if they added any features that might help with the process of HOG detection, but the newer version might have a better implementation.

5.2 Sensor Fusion

Low-cost solutions exist which can achieve an accuracy $\sim$ 1cm in geo-location, most notably the NAVIO with RTKLib. Fusing information from RTK gaps with IMU, as well as the data from LSD-SLAM shows promise of a highly accurate low-cost solution.

5.3 Feature Localization and Tracking

Feature localization can be improved by detecting SURF/SIFT features within the detection windows. We can then store these features in a data structure to be matched by future frames. With slow moving objects, such as humans, the movement from frame to frame will be rather small. With a frame rate of 30fps, assuming a human walks at 5 mph we can assume a translation in world coordinates of less than 100cm per frame. With this small change, we can also triangulate the objects distance from the camera. This added information will eliminate the issue of having to assume the feature we are tracing is on a flat plane, increasing the accuracy of the location significantly. Furthermore, by having a library of detected feature points for every object, it is plausible that it will be possible to reacquire an object after it moves off frame, then back into the frame.

5.4 Occlusion and Skipped Detection Windows

Much work has been done "filling in" missing data. Using a Kalman filter or similar algorithm, it is possible to continue tracking an object when the HOG feature detector fails to detect a previously detected object to occlusion, or failure.

5.5 Real-Time Aplication of Human Detection

At the moment, most CPU-based HOG algorithm implementations takes seconds to process a frame at 1920x1080 resolution from our results. With usage of a GPU based algorithm though, the algorithm run fast enough to process a low quality resolution video in real-time. This was tested out on an NVidia 650m graphics card, so with a high quality, modern graphics card, it may be possible to process an HD video in real time. Another possibilty is to use a embedded board with gpu capabilities for processing data in real time on the unmmanned aerial platform. Finally, moving to a language that is suited for runtime applications, such as C++ would halp as well with speeding up the HOG algorithm.

5.6 Conclusion

The results from the confusion matrix and classification report were as expected for detecting humans. With the models from the k-fold cross validation test, the average from the model's scores was a rate of 0.81 accuracy.The classification report reports values that fall in the range of other scholarly articles discussing human detecton rates with HOG. We believe that these results were achieved by the larger dataset giving a more varied sample of positive and negative images.

Average of K-Fold Cross Validation Model Scores **0.81**

Confusion Matrix

$$\begin{bmatrix} 292 & 83 \\ 45 & 337 \end{bmatrix}$$

Class. Report

	Prec.	*Recall*	$f1score$	*Support*
$NonHuman$	0.87	0.78	0.82	375
$Human$	0.80	0.88	0.84	382
$\frac{avg}{total}$	0.85	0.83	0.82	1136

Unfortunately, our real world application results were sub-par. The video would move at the speed of approximately one frame per minute, with false positive detection of humans in each frame, for the two classifiers created by us. Even the default classifier had runtime issues, with it running about 1 frame per second. Since the results were reasonable from the initial SVM training/testing, we speculated that the parameters for either creating the SVM model, or the paramters for the $detectMultiScale()$ method were causing the issue. After fidgeting with the parameters to no avail, we had to resort to the default classifier for further application usage.

The results from the data set showed the use of projected coordinates are superior to using overlapping windows for matching same features when the camera is not in a fixed position.

References

Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*, CVPR '05, pages 886–893, Washington, DC, USA, 2005. IEEE Computer Society.

Jakob Engel, Thomas Schöps, and Daniel Cremers. Lsd-slam: Large-scale direct monocular slam. In *Computer Vision–ECCV 2014*, pages 834–849. Springer International Publishing, 2014.