Salamander: computer vision for wildlife research

Christopher Patton

March 9, 2013

Abstract

This document outlines an image prcessing application called Salamander. Salamander is comprised of a set of algorithms for automatically detecting and tracking targets of interest in a video stream. It is particularly useful for filtering and segmenting video produced by field cameras for wildlife and behaviorial research.

Contents

1 Introduction

[TODO] Overview of approach. Define each task: detection, tracking, and segmentaiton.

2 Algorithms

I now describe our approach to target detection and tracking and segmentation of the video stream. We make the following basic assumptions about the video and our targets:

1. The camera is fixed.
2. The rate at which pictures are taken is fixed.[1]
3. The targets are the same species, i.e., roughly the sames size and color. Under typical circumstances, its not practical to look for deer and mice simultaneously.
4. Targets are roughly the same distance away from the camera when we detect them.

The reasons for these assumptions will become apparent in this section. They are necessary for the basic functionality of the algorithms we'll describe, but it may be possible to disregard at (3) and (4) with machine learning. We'll explore this possibility in section 3. However, we must assume (1) in any case.

Initial image processing, detection, tracking, and segmentation are layered. By this we mean that in order to achieve one task, all preceeding tasks must be accomplished first. We describe each layer and the corresponding algorithms and data structures in turn. In the following sections, a video stream is represented by a sequence of video frames:

$$\langle v_0, v_1, v_2, \ldots v_{n-1}, v_n \rangle$$

2.1 Image processing

The first step in our approach is to calculate the pixelwise difference of two adjacent frames in a video stream. The resulting image, which we'll refer to as the *delta* image, tells us which regions of the image changed within that unit of time. We'll perfom some additional filtering, then search the image for *blobs*, contiguous regions of *delta* that correspond to significant changes. At this point, image processing is finished and we can proceed to detection.

delta $\delta(v_{i-1}, v_i) \rightarrow d_i$. Input images are typically JPEG format. Each pixel of a JPEG image is represented by three bytes corresponding to the red, blue,

[1] It may be necessary in certain implementations to covercome this assumption. For example, a field camera may transmit its video stream over a network. The rate at which pictures are taken can be set to depend on network conditions. However, It is possible to assume a lowest-possible rate and tune Salamander accordingly.

and green luminance. To simplify the following steps of the pipeline, we first convert v_{i-1} and v_i to gray scale images with floating point precision. Next we subtract v_i from v_{i-1} and store the absolute value of each pixel. Now, all changes in the frame within the time points $i-1$ to i are reflected in the delta image d_i. We can analyse these changes in order to determine if a target has appeared in frame.

[THINK ABOUT THIS] It's true that we lose a bit of information when we apply abs filter. If the target is lighter than the background, then it's old position will be negative in *delta* and the new position will be positive. However, the reverse is true if it is darker than the background. Perhaps we can remember the relative darkness of the target when it first appears.

binary threshold $b(d_i) \rightarrow b_i$. The next step converts all pixels to one of two values {white, black}, which is needed for subsequent filters. This step also allows us to deal with noise artifacts related to the quality of the image taken, e.g. bluriness, compressing and decompressing JPEGs, gray scale filtering, etc.[**ref**] More to the point, if a pixel has changed just a little, we set it to black; if it's changed a lot, we set it to white. For each pixel p in d_i and threshold range l and h where $l < h$,

$$b_i(p) = \begin{cases} \text{white} & l \leq d_i(p) \leq h \\ \text{black} & \text{otherwise.} \end{cases}$$

[TDOD] experiment with upper threshold higher than 255? Since floating points in ImageType can be higher I'm pretty sure.

binary morphology $m(b_i) \rightarrow m_i$. finally, the last filter enables us to deal with physical noise. This includes things like shadows, small objects such as leaves or grass being blown around by the wind, and small insects we may not care about. Morphology has two stages: in erosion, the edges of artifacts in b_i are worn away iteratively. Small noise artifacts will disappear as a result, leaving only those that correspond to actual targets; in dilation, we inflate the remaining blobs by doing the reverse of erosion. We iteratively add layers to the surfaces of remaining blobs. Typically, many more dilation as erosion rounds are used, so that artifacts that comprise a single target, i.e., those that are near to each other, will merge together. Binary morphology is the most computationally expensive stage of the image processing pipeline. Once we're done, we're ready to analyse the remaining *blobs* for the presence of targets.

connected component analysis $c(m_i) \rightarrow$ *Blob list*. The final image processing step allows us to disvoer the locations of blobs in m_i and provides some useful information about them. The connected component analysis algorithm, analogous to its counter-part graph theory, returns a list of connected regions in the image. In the binary image m_i, this is simply contiguous regions of the same color. The black background is one such region; the rest are the blobs remaining after binary morphology. Let $c(m_i) :\rightarrow$ *Blob list* be a function that

performs this analysis and returns a list of blobs with the following attributes:

$$Blob = \begin{cases} \textit{bbox}(l,r,t,b) \texttt{ int} & \text{Rectangular region bounding the blob.} \\ \textit{centr}(x,y) \texttt{ int} & \text{Coordinates of blob centroid.} \\ \textit{vol} \texttt{ int} & \text{The number of pixels that comprise the blob.} \\ \textit{elong} \texttt{ float} & \text{Elongation, the longest straight line that traverses the blob.} \end{cases}$$

The *Blob.bbox* tuple gives the coordinates of the bounding box. E.g., (l, t) refers to the top-left corner, (r, b) the bottom-right, etc. The bounding box is central for our tracking methodology, The other features are used in various ways for target detection, as we'll see in the next section.

2.2 Detection

At the moment, we employ a very simple detection strategy. We first assume that one blob corresponds to a single target, i.e., all blobs comprising a single target have merged as a result of dilation in binary morphology. Then, if any blob in *delta* has a volume larger than a given threshold, we say the frame contains a target:

func *detect*$(V \equiv \langle v_0, v_1, v_2, \ldots v_{n-1}, v_n \rangle, n) \rightarrow D \subset V$
    $D = \emptyset$
    for $i \leftarrow 1$ **to** n **do**
        for *blob* **in** $c(m(b(\delta(v_i, v_{i-1}))))$ **do**
            if *blob.vol* $\geq$ *thresh* **then**
                $D = D \bigcup v_i$
                break
            fi
        done
    done
    return D
end *detect*

With small targets, e.g., mice, rats, and salamanders, this scheme has proven sufficient. However, for larger targets, and for the sake of generality, a better scheme is necessary.

2.2.1 A more sophisticated scheme

[THINK ABOUT THIS IN JUNE] I think a better method would be k-clustering with euclidean distance. In general we've been using $e << d$ morphology factors, assuming that all blobs corresponding to a target would merge as a result of dilation. In addition, we've assumed that all noise is squelched in the image processing pipeline. These are bad assumptions in general and will break entirely with bigger targets, e.g. deer.

A better idea is to use $e = d$ and consider the distribution of blobs. Regions of dense blob are likely to correspond to a target in the frame, where blobs in

less dense areas are likely noise. We can use k-clustering (or k-centroids?) to identify groups of blobs that comprise a target. We then use the bounding box of all these blobs for tracking.

Of course, k-clustering for any k is NP-hard, so we need to introduce a heuristic for [TODO] I should run some tets with the method to see how affective it is in practice.

2.3 Tracking

Shifting over merged blobs. Multiple targets.

2.4 Segmentation

Data structures. Testing if gap has target.

3 Machine learning

[FUTURE] I still need to research how and if to do this.

4 The Salamander toolkit

Salamander is a C++ extension to the Insight Registration and Segmentation Toolkit[2]. ITK provides a comprehensive array of image processing algorithms and is designed to be useful in many applications. A key design feature of ITK is its support of generic programming. As well as processing two-dimensional images, it can process multidimensional image and mesh data, making it a successful framework for medical image analysis. This architecture is extremely useful when, for example, a project must adapt and scale to new data formats. However, in our computer vision application, we've restricted ourselves to two dimensinal images and affectively ignore the vast majority of ITK's featurs. As a result, we encurr the overhead of generic programming without gaining its benifits.

Therefore, it may be prudent to switch to a framework that is centered around computer vision. We will consider the feasibility of using OpenCV[3] instead of ITK. OpenCV is written in C and doesn't support generic programming. As long as the image processing algorithms we need (pixelwise subtraction, binary thresholding and morphology, connected component analysis) are comparably efficient, OpenCV will be a viable replacement.

Another advantage of OpenCV over ITK is that it provides a well-known means of implementing machine learning, something that we'll explore in future iterations of the Salamander project.

[2]ITK is an open source product by Kitware and is available at `http://www.itk.org`.

[3]`http://opencv.org`.

4.1 Build and install

Both ITK and Salamander should be built from source. ITK is available for download at `http://www.itk.org/ITK/resources/software.html`. As of this writing, Salamander is hosted on github at `http://www.github.com/cjpatton/salamander`. The build process for both is identical, as they use `cmake`. In the source directory, under GNU/Linux:

```
$ mkdir build && cd build
$ cmake ../
$ make
$ sudo make install
$ sudo ldconfig
```

ITK has a large amount of code and will require a couple hours to finish. As always, two cores can be assigned to building with `make -j 3`.

[TODO] Windows instructions.

4.2 A quick example

If you've successfully installed Salamander, you can try this tutorial to see how it works. In the source directory, go to `ex/three`. This footage is from an experiment with mice, in which they are the choice of two environments. The goal wsa to find out which they prefer. Type the following at the terminal:

```
$ ls video*.jpg | segment -t 20 60 -m 1 15
```

The program will output the chunks of video in which a mouse appears in frame. Also, a series of images are produced called `tracking*.jpg`, in which `segment` has drawn bounding boxes around the location of the target.

4.3 Usage

After installation, there are a number of top-level programs available. This section documents what they do and how to use each of them. Each Salamander program accepts a list of filenames on standard input and sorts thenm alphanumerically to ensure they are processed in the correct order. Except for `binthresh` and `binmorph`, all of the programs have the same command line options which enable tuning of the image processing parameters. These are:

`-t` l, h Binary threshold range, where $0 \leq l \leq h \leq 255$. Any pixel whose value falls between l and h become white: otherwise, black. Although we can make adjustments to reduce noise here, this value should be set so that an interesting blob appears white after this stage. Therefore, a wide filter is commonly used, such as $l = 20$ and $h = 60$.

`-m` e, d Binary morphology factors. e specifies the number of erosion iterations and d specifies the number of dilation rounds. Typically, $e < d$. There are two considerations for this parameter: one, the relative size of the target, and two, the frequency of noise.

`-s` s Image shrink factor, where $s > 0$. Image processing is far and away the bottleneck of Salamander. We can increase throughput by shrinking the image. The trade off is that we can lose information and, potentially, miss small targets. If $s = 2$, then the image is shrunk by a factor of two. Default is $s = 1$ (no shrinking).

`-f prefix` Output file prefix. Some programs output files.

4.3.1 `binthresh` and `binmorph`

These simple programs apply image processing filters to the images and output the result as a sequence of images. They are useful for experimenting with parameters on a small set of images. `binthresh` accepts two arguments corresponding to the binary threshold range. If none are provided, only the image difference filter is applied. `binmorph` accepts four arguments corresponding to the morphology factors and the threshold range. If two are provided, these are assumed to be the morphology factors and the threshold defaults to $l = 20$ and $h = 60$.

E.g.: `$ ls video*.jpg | binmorph 1 10 20 60`

4.3.2 `filter`

This program is functionally similar to `binmorph` except that it provides all the normal tuning parameters and provides more useful information. In particular, for each delta image, it outputs a list of blobs. `filter` is useful for testing parameters over a large set of images. In the future, it will be modified to be useful for collecting statistics [IMPLEMENT].

E.g.: `$ ls video*.jpg | filter -t 20 60 -m 20 60`

4.3.3 `detect`

Currently, this program is being used to test the more advanced detection scheme discussed in `2.2`. [IMPLEMENT] Once this is finished, this program will simply output the images that contain targets.

4.3.4 `segment`

Currently, `segment` is the highest level program. It applies the image processing filters, detects the presence of targets, tracks the target in time, and outputs segments of video that contain targets. [TODO] Rename files by segment number, dump to disk. Delete all files that don't appear in a chunk.

E.g.: `$ ls video*.jpg | segment -s 3 -t 20 60 -m 20 60`

4.4 Processing a feed in realtime

[NOT IMPLEMENTED] This feature should be implemented as a flag at the command line.

4.5 Code documentation

[TODO] I hope to generate documentation with doxygen. This will be plain HTML.