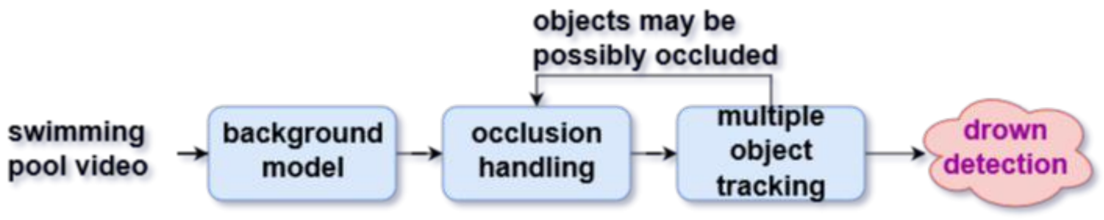
Automatic Drowning Detection Surveillance System

Instructor: Jing-Jia Liou   Members: Ying-Yi Liao, Johnson Wang, Yang-Shi Su

Abstract:

Drown detection system aims to assist lifeguards with swimming pool surveillance tasks. Each year, it is impossible to end a summer without hearing a single news of people drowning in swimming pools, rivers or beaches. Even under the surveillance of several well-trained lifeguards, the environment in the water changes so rapidly that it requires highly concentration for lifeguards to keep track of every motion in a certain area. However, a blink of an eye is all it needs to take for an accident happen in a water area. Therefore, with a device that won't feel tired as long as the power is on, we could not only have a comprehensive view about what is going on in the area, but also, most importantly, save lives.

In our project, we take a small taste on drown detection system. We have worked on both software and hardware part. For software part, the goal is to detect whether there is a drown swimmer. In our implementation, it can be basically separated into four part, shown as follows,



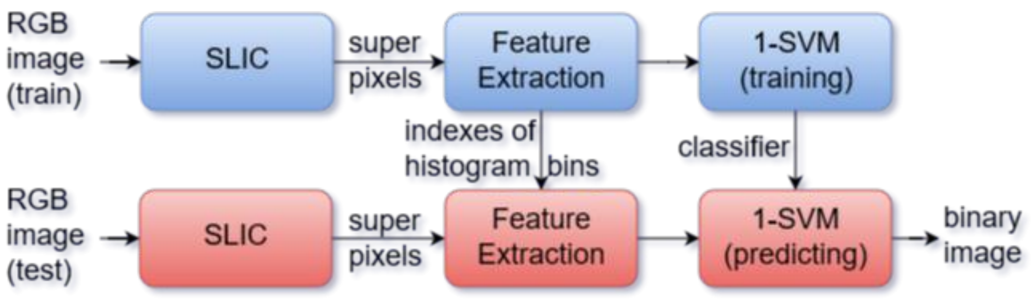
The last part, which is also, in some degree, the most critical part, drown detection, is unfortunately undone. We did take efforts to extract features used for the training process. However, lack of good training data is a great obstacle in the beginning of this section. We then temporally gave up this task and turned to other parts. An above-water surveillance video of a swimming pool is converted to a foreground and background binary image through background model. Then, occlusion handling process is performed if the latter multiple object tracking process finds out that there may be occluded objects. Subsequently, all objects are kept in tracks, recorded, and extract features for upcoming drown detection process. For hardware part, we use Webcam Logitech C310 to capture frames and Raspberry Pi 2 for data collection. Each Raspberry Pi 2 takes charge of one Webcam, which is used as a local client, collecting frames captured by the Webcam and sends to the remote central server. Receiving frames from every spots where we place Webcam, then it is possible to do the

subsequent processing steps by our central server.

1. Background model:

   In order to detect swimmers (objects) who have high potential to be drown, a preprocessing work that segments background from foreground, which may be all swimmers or objects in the pool, should be done. My background model is primarily based on novelty detection and can be basically divided into two part, training and predicting. In training part, videos of a swimming pool with no swimmers or any other objects are inputs of training process, outputting a classifier that learn “what is a swimming pool”. In predicting part, the classifier just mentioned will try to discriminate “what is a swimming pool”, i.e. background, through videos of a regular swimming pool, and for those that the classifier think they are not a swimming pool will be labeled as foreground. The final output of the background model will be a binary image segmenting background from foreground.

   The main motivation of this framework is that I think this is user-friendly and convenient, since novelty detection allows users simply run training process when the swimming pool is closed (with no swimmers in the swimming pool), requiring no extra manipulation over the system. A more specific view of my background model is shown as below.



   A frame, which may be a RGB image, of a video of a swimming pool with no swimmers is first segmented into numerous superpixels using SLIC (simple linear iterative clustering) algorithm. Subsequently, local image features are extracted from every superpixel. 1-SVM classifier is then trained with those features. After training process, a frame of a regular swimming pool video go through the same process in training part, which are superpixel segmentation and feature extraction. Finally, 1-SVM classifier will detect novelty, which will be labeled as foreground, within all superpixels, and output a binary image distinguishing foreground from background.

   - SLIC (simple linear iterative clustering):

     Superpixels are a popular technique used in computer vision, and SLIC algorithm is a method among many others that produces superpixels with high quality segmentation, consistent size, and explicit control of number and compactness of superpixels. Superpixels provide a convenient and a way more efficient primitive from which to compute local image features. They inherently eliminate redundancy of an image, e.g. in most time, information in small neighborhoods is good enough, making per pixel information unnecessary, and therefore greatly reduce the complexity of subsequent tasks, e.g. a 1024×1024 frame may produce 1048576 feature vectors, but with superpixels, 5000 superpixels, implying 5000 feature vectors, are good enough, resulting in far less computational cost for, in our case, 1-SVM training process.

SLIC performs a local clustering of pixels in 5D space defined by L, a, b value in CIELab color space and x, y coordinate of pixels. Clustering pixels, where every cluster represents a superpixel, is based on color similarity and proximity in the image plane. SLIC algorithm can be easily comprehended in view of 3 parts, distance measure, updating clusters' centers, and iterative process.

1. Distance measure:

   Distance measure specifies the distance computed between two points in 5D space. CIELab color space is widely known as possessing perceptually uniformity for small color distances, i.e. with no drastic change, distance of two color vectors in CIRLab color space is proportional to subjective feeling of human vision. While the maximum distance between two colors in CIELab color space is limited, the spatial distance between two pixels is based on image size. We cannot directly use Euclidean distance in 5D space. A new distance measure comes into rescue by normalizing the spatial distance. Using it, color similarity as well as pixel proximity in 5D space are both taken into consideration such that expected cluster size and their spatial extent are approximately equal.

   A new distance measure $\mathrm{D_s}$ is defined as follows:

$$\mathrm{d_{Lab}} = \|c_1 - c_2\|_2, c \in color\ vectors\ in\ CIELab\ space$$
$$\mathrm{d_{xy}} = \|p_1 - p_2\|_2, p \in locations\ in\ xy\ coordinate$$
$$\mathrm{D_s} = \mathrm{d_{Lab}} + \frac{m}{S} d_{xy},$$

   where m is a coefficient allowing as to explicitly control the compactness of a superpixel, and S is a constant precomputed based on the number of superpixels determined by users. For a desired number of approximately equal-sized superpixels K and an image with N pixels, the approximate size of every superpixel is $\frac{N}{K}$ pixels. Roughly equal-sized superpixels may have their centers at grid interval $S = \sqrt{\frac{N}{K}}$.

2. Updating clusters' centers:

   Begin with K centers at regularly-spaced grid interval S (mentioned in upcoming section) and move them to seed locations corresponding to the lowest gradient in a 3×3 neighborhood.

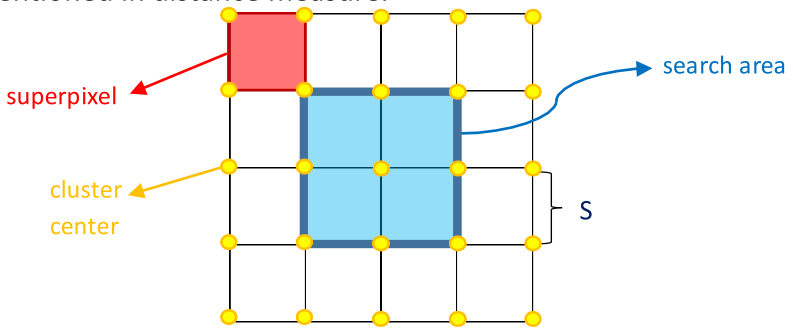
$$\underset{x,y}{\operatorname{argmin}} \|I(x+1,y)-I(x-1,y)\|_2^2 + \|I(x,y+1)-I(x,y-1)\|_2^2$$

   where $I(x,y)$ is the Lab color corresponding to the pixel at position (x,y), and $\|\cdot\|_2$ is L2 norm. This is done to avoid placing initial cluster centers at discontinuity of the image, i.e. at an edge or a noisy pixel.

   The aforementioned are the initial setting of cluster centers. Every time after all pixels are associated with the nearest cluster centers, centers are updated as the average (or in k-means algorithm's sense, centroids) of $Labxy$ vectors of all pixels belonging to the cluster.

3. Iterative process:

   In the beginning of SLIC algorithm, we choose K cluster centers $C_k = [L_k, a_k, b_k, x_k, y_k]^T, k \in [1, K]$ at regular grid interval $S = \sqrt{\frac{N}{K}}$, mentioned in distance measure.



   As shown in above graph (regular grid used for demonstration), the spatial extent of any superpixels is roughly S^2, and thus we can safely assume that pixels assoiciated with this cluster center lie within a $2S\times2S$ area around the superpixel center on the xy plane, i.e. for those pixels out of bounds, they will surely belong to other superpixel centers rather than the center aforementioned. Note that computing distance from each cluster center to pixels within a $2S\times2S$ region implies a much smaller search area, in comparison to conventional k-means algorithm, where distance are computed from each cluster center to every pixel in the image.

   SLIC (simple linear iterative clustering) is summarized as follows.

Algorithm 1 Efficient superpixel segmentation

```
1: Initialize cluster centers $C_k = [l_k, a_k, b_k, x_k, y_k]^T$ by sampling pixels at regular grid
   steps $S$.
2: Perturb cluster centers in an $n \times n$ neighborhood, to the lowest gradient position.
3: repeat
4:    for each cluster center $C_k$ do
5:       Assign the best matching pixels from a $2S \times 2S$ square neighborhood around
         the cluster center according to the distance measure (Eq. 1).
6:    end for
7:    Compute new cluster centers and residual error $E$ {$L1$ distance between previous
      centers and recomputed centers}
8: until $E \leq$ threshold
9: Enforce connectivity.
```

At the end of above process, a few stray labels may remain, i.e. pixels in the vicinity of a segment having the same label are not directly connected to it. This is rare but can possibly happen since spatial proximity clustering does not explicitly enforce connectivity (do not forget distance measure also features Lab color distance). Accordingly, Connectivity enforcement is done in the last step of the algorithm, relabeling disjoint segments with the labels of the largest neighboring cluster.

SLIC can be, in some degree, viewed as a different version of k-means algorithm. Basically, both of them go through similar process, associating pixels to cluster centers, updating cluster centers, and repeat until converge. The primary distinction are distance measure (computed in determining which cluster centers a pixel belongs to), cluster centers updates, and search area.

- Feature extraction:

RGB images are first converted to CIELab color space. We only take values of a and b dimension (color-opponent dimension) into consideration, since L dimension represents lightness and does not determine colors. Acting as input of following 1-SVM, feature vectors are extracted per superpixel.

$$f_{a,k} = primary(h_a/\max(h_a))$$

$$f_{b,k} = primary(h_b/\max(h_b))$$

$$f_k = [f_a \;\; f_b], k = [1, K], K\ is\ the\ number\ of\ superpixel$$

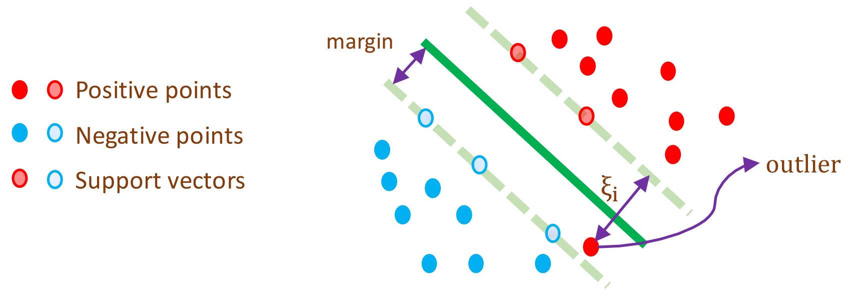
Histograms with 16 bins over a and b dimension are calculated in a superpixel, denoted h_a and h_b. $\frac{1}{\max(h)}$ roles as a normalization factor for superpixels have "approximately consistent" size and thus we need to compensate for the slight deviation ($\sum h$ does not solve the problem, and that is why I use $\max(h)$). Furthermore, the purpose of $primary$ function is dimension reduction. In my implementation, it reduce the original 32 dimensions feature vectors to 5 dimensions. It is simply a thresholding function which preserves non-zero bins (indexes of preserved bins are determined in training process and in predicting process, they are fixed.). Combination of normalized a and b dimension histogram describes the color of water, and therefore will cluster together in feature space. For this reason, PCA (principle component analysis) is not suitable. Or to be more general, I think PCA is not suitable for novelty detection. Finally, concatenate two reduced and normalized histograms and we get the feature vector of a superpixel.

- 1-SVM (one class support vector machine):

  1-SVM is an algorithm designed for solving one-class classification problems. Unlike conventional classification problems, which may have training data belongs to multiple classes and classify new test data into those classes, 1-SVM determine whether new test data is member of a specific class, determined by training data. In other word, 1-SVM detects the abnormal based on the knowledge of training data, otherwise known as novelty detection. In our case, videos of an unmanned swimming pool (to be more precise, water) are normal training data, and any object that is not swimming pool (or water) is labeled as out-of-class and considered as foreground.

  ✓ Basic concept of Support Vector machine:

  We first look at the most basic two-class SVM. Given m data points $\{(x_1, y_1), (x_2, y_2) \dots (x_m, y_m)\}$, where x_i can be an arbitrary dimensional vector and $y_i \in \{-1, +1\}$ is the label of i^th data points. Input vectors are first projected by function $\Phi(\cdot)$ to a higher dimensional feature space H, forming the same number *m* of feature vectors (established in next section). Those feature vectors will be separated by a hyperplane $w^\mathrm{T}\Phi(x) + \mathrm{b} = 0$, where $w, \Phi(x) \in \mathrm{H}$, $\mathrm{b} \in \boldsymbol{R}$, and the decision function on a new point to be predicted is $f(new\_x) = sgn(w^\mathrm{T}\Phi(new\_x) + \mathrm{b})$.



The desired hyperplane is constructed through maximizing the margin ρ. Margin is computed using geometric distance between hyperplane and the closest point to the hyperplane.

$$\rho = \min_{i \in [1,m]} \frac{|w^\mathrm{T}\Phi(x_i) + \mathrm{b}|}{\|w\|}$$

For mathematical convenience, the hyperplane is defined in canonical representation such that $\min_{i\in[1,m]} |w^{\mathrm{T}}\Phi(x_i) + \mathrm{b}| = 1$. After defining the canonical representation, two hyperplanes $w^{\mathrm{T}}\Phi(x) + \mathrm{b} = \pm 1$ are formed associated with margin of hyperplane $w^{\mathrm{T}}\Phi(x) + \mathrm{b} = 0$, so-called marginal hyperplanes. Thus, marginal maximization can be formulated as $\max_{w} \frac{1}{\|w\|}$.

The objective function of two-class SVM classifier is the following,

$$\min_{w,b,\xi_i} \frac{\|w\|^2}{2} + C\sum_{i=1}^{m} \xi_i$$

$$subject\ to\colon \quad y_i(w^T\Phi(x_i) + b) \geq 1 - \xi_i, \qquad \forall i \in [1, m]$$
$$\xi_i \geq 0, \qquad \forall i \in [1, m]$$

The first term of minimization formulation enforces marginal maximization, and slack variable ξ_i (visualized in above graph) in the second term allows certain violation over hyperplane separation on the point $\Phi(x_i)$ in H. Slack variable ξ_i is introduced to prevent SVM classifier from over-fitting noisy data, creating a soft margin (having tolerance on outliers). The constant C is a free parameter that determines the trade-off between maximizing the margin and the number of outliers, i.e. training errors.

The minimization problem with quadratic programming can be reformulated as Lagrangian dual function and solved using some optimization method, such as SMO. The decision function then becomes

$$f(new\_x) = sgn(\sum_{i=1}^{m} \alpha_i y_i K(new\_x, x_i) + b)$$

where α_i are the Lagrange multiplier, acting as weights of feature vectors projected from training points. Those x_i with non-zero α_i are called support vectors for they “support” the construction of the separating hyperplane. Often, SVMs are considered to be sparse, which means there will be relatively few Lagrange multipliers with non-zero value, bringing huge advantage to exhaustive-search-like optimization method.

✓ Kernel method:

A kernel function is defined as $K\colon \mathrm{X}\times\mathrm{X} \rightarrow \boldsymbol{R}$, where X is input space.

$$\forall x_1, x_2 \in \mathrm{X}, \qquad K(x_1, x_2) = < \Phi(x_1), \Phi(x_2) >$$

$<\cdot,\cdot>$ is inner product. Function $\Phi(\cdot)$ is a mapping from X to a higher dimensional feature space H, and so-called feature mapping. As mentioned in the above, SVM is a linear classifier, and it cannot well handle non-linear separable data points. This is where kernel trick comes into rescue. It allows projection from input space with no hyperplane having ability to separate data points to a higher dimensional feature space, which can be of unlimited dimension and thus the hyperplane can possibly separate very complex data.

Kernel method is extremely powerful in SVM, since outcome of decision function of SVM only relies on the dot-product of vectors in the feature space, implying that we do not need to explicitly define the feature mapping function $\Phi(\cdot)$. As long as kernel function K can be correctly computed, it is good enough. Popular choices for kernel function are linear, polynomial, sigmoidal, and Gaussian radial base function (RBF). RBF kernel is used in my implementation.

$$K(x_1, x_2) = e^{-\frac{\|x_1 - x_2\|^2}{2\sigma^2}}$$

where $\sigma \in \mathbf{R}$ is a RBF kernel parameter and $\|\cdot\|$ is distance measure.

✓ One-class SVM:

We first define a set of separable data points $x_i \in X, i \in [1, m]$, where X is input space as,

$$\exists w \in X, b \in \boldsymbol{R}, w^T x_i + b > 0, \qquad \forall i \in [1, m].$$

It implies a simple assumption that the set of data points cluster compactly in the input space. One-class SVM deals with problems having input data like that. It returns a classifier f that takes the value +1 in a "small" region capturing most of training data points, and -1 elsewhere. The basic concept of one-class SVM is to first map training data into higher dimensional feature space H and to separate them from the origin with maximum margin. Freedom of utilizing different kernel functions provides a variety of nonlinear estimators in input space. To separate the data set from the origin, the objective function is given as follows,

$$\min_{w,\xi_i,\rho} \frac{\|w\|^2}{2} + \frac{1}{\nu m}\sum_{i=1}^{m} \xi_i - \rho$$

$$\textit{subject to}: \quad w^T \Phi(x_i) \geq \rho - \xi_i, \qquad \forall i \in [1, m]$$

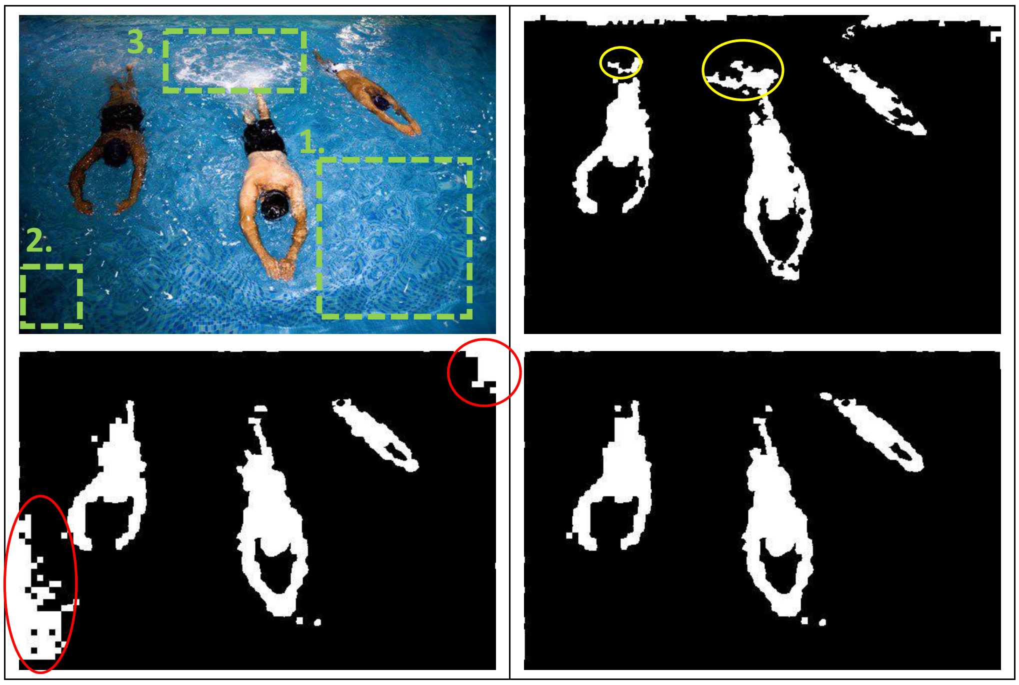
$$\xi_i \geq 0, \qquad \forall i \in [1, m]$$

Here, $\nu \in (0,1)$ is a free parameter that sets on the upper bound on the fraction of outliers (training points classified as -1, or out-of-class) and a lower bound on the number of training points taken as support vectors. The decision function with Lagrange multiplier introduced is,

$$f(new\_x) = sgn(\sum_{i=1}^{m} \alpha_i y_i K(new\_x, x_i) - \rho)$$

A hyperplane characterized by $\boldsymbol{\alpha} \in \boldsymbol{R}^m$ and ρ separates a compact set of data points from origin in feature space, returning -1 for any "out-of-class" points in test data.

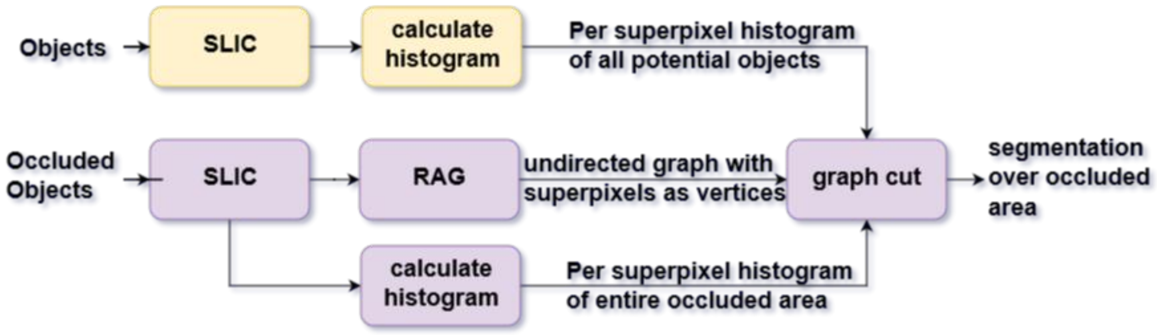
- Experimental result:



The left-upper image is the input of our background model, and the rest of them are outputs using different training data for 1-SVM classifier (all output images are filtered by contour area to suppress noise). We first look at the right-upper output image. This is classified by using normal water image (shown as 1. and 2. rectangle in the input image). We can find that water splashes with extreme lightness cannot be well-separated from swimmers. Besides, we need to also consider dark area of water region (shown as 2. Rectangle in the input image). Otherwise, the output will be like the left-lower image (with training data as 1. and 3. rectangle in the input image). Hence, with overall consideration, we can get a much better result shown as the right-lower image, in which water splashes and the shadowed area are well-handled.

2. Partial occlusion handling:

When we have obtained foreground and background binary image, the next job is to keep track on every existing objects (swimmers). However, there are some times that partial occlusion happens among several objects, i.e. an object is partially covered by another object. Accordingly, occlusion handling is of great importance so as to avoid incorrect object tracking, which is based on finding contours of objects (established in the next section). The overall procedure of my occlusion handling method is shown as below,



Objects
SLIC
calculate histogram
Per superpixel histogram of all potential objects
Occluded Objects
SLIC
RAG
undirected graph with superpixels as vertices
graph cut
segmentation over occluded area
calculate histogram
Per superpixel histogram of entire occluded area

Initially, objects that are potentially one of the members of an occluded area are over-segmented into numerous superpixels by SLIC algorithm (mentioned before), and we compute histograms for every superpixels in every potential objects, denoted as $H_i^{(k)}, i \in [1, N], k \in [1, M]$, where N is the number of superpixels and M is the number of potential objects. Subsequently, occluded objects where we aim to discriminate an object from the others go through the same process, getting per superpixel histogram $H_i', i \in [1, N]$. The use of features computed in over-segmented area gives more precise information, in comparison to pixel-wise feature, which may be misleading, e.g. in the case of two similar color objects. Furthermore, an undirected graph consisting of superpixels as vertices is obtained by performing RAG (Region Adjacency Graph). Next, an energy function is formed based on per superpixel histogram of all potential objects and occluded area, and the graph structure of occluded area. The energy function then can be minimized through graph cut algorithm, giving a structural prediction over labels of all superpixels in occluded area. Finally, we can get a labeled image with appropriate segmentation of every occluded objects.

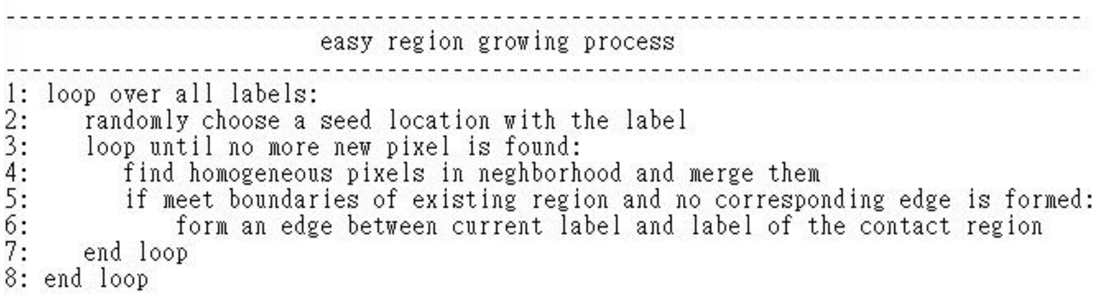
- RAG (Region Adjacency Graph):

  In order to construct a graph (SLIC does not give edges) as an input of the following graph cut, region growing process (RGP) is performed to obtain region adjacency graph, i.e. every region corresponds to a vertex and constitutes homogeneous pixels, where homogeneity can be self-defined. In our case, finding RAG may be the simplest task among other implementation, e.g. watershed process used in image segmentation, since inputs are labels image featuring superpixels. This brings an easy definition for homogeneity, and that is pixels with the same label are considered to be in the same region.

The most basic ideas of region growing process for constructing a RAG can be described as follows. Let X be all pixels of an image, and P be a logical predicate defined on a set of contiguous pixels. Then we define disjoint and non-empty subset $R_i,\ i \in [1, N]$ assembling the whole image X such that,

1. $\bigcup_{i=1}^{N} R_i = X$
2. $R_i, i \in [1, N]\ is\ connected$
3. $P(R_i) = True, \forall i \in [1, N]$
4. $P(R_i \cup R_j) = False, \forall i, j \in [1, N], i \neq j,\ R_i\ and\ R_j\ are\ adjacent$

Note that predicate P determines homogeneity, like similar color distribution and in our case, the same label. The second condition implies that all pixels in the same region must be connected. This constraint is inherited in the region growing process for the "flood-like" process to merge pixels is carried out over neighborhood pixels. The third and fourth criteria enforce similarity in the same region and discrepancy between adjacent regions. The naïve way of the above process can be shown as follows,



```
--------------------------------------------------------------------------------
                         easy region growing process
--------------------------------------------------------------------------------
loop over all labels:
   randomly choose a seed location with the label
   loop until no more new pixel is found:
      find homogeneous pixels in neghborhood and merge them
      if meet boundaries of existing region and no corresponding edge is formed:
          form an edge between current label and label of the contact region
   end loop
end loop
```

Again, the above algorithm is only the easiest version of RGP, which is good enough for our case. In common cases, estimate of homogeneity may be a more sophisticate mathematics formula and the growing process may not be so straightforward, e.g. non-sequential growing, or de-growing stage to obtain optimal region. Finally, we can get our result with all adjacent superpixels connected by edges.

- Graph cut:

Graph cut algorithm is a popular method used in low-level computer vision, such as stereo matching, image smoothing, 3D reconstruction, image segmentation, and other problems which can be formulated as energy minimization. The most classic use of graph cut is to solve pixel-labelling problem. From my perspective, the greatest advantage using graph cut is that through formulating valid energy function, we can obtain structural prediction, which means it takes structure, or for images, spatial information, into consideration, for spatial continuity over pixel-wise features often holds in an image except for edges. Furthermore, edges can be dealt with by introducing spatial cues to our energy function.

- ✓ Energy function:

The standard form of the energy function is shown as follows. Given a set of pixels P and a set of all possible labels L to which pixels may be assigned. We aim to find a labeling f (a mapping from P to L) such that the following objective function is minimized.

$$E(f) = \sum_{p \in P} D_p(f_p) + \lambda \sum_{(p,q) \in Nb} V_{p,q}(f_p, f_q)$$

where $Nb \in P \times P$ is the neighborhood system on pixels. $D_p(\cdot)$ is data cost function on each individual pixels, i.e. it specifies the cost of assigning certain label to a pixel. $V_{p,q}(\cdot,\cdot)$ is an interactive term that specifies relationship of pixels in a neighborhood, measuring the cost of assigning labels to two adjacent pixels, and can be viewed as smoothness term. $V_{p,q}$ has great strength for it inherits spatial information of an image and brings a more overall view of an image rather than that derived from an individual pixel. Finally, λ is a coefficient that makes a trade-off between data cost D_p and smoothness cost $V_{p,q}$.

The following describes the formulation of my energy function. Label space is defined on the number of objects that have potential to be in the occluded area, i.e. $L \in [1, M]$. Recall the symbols of per superpixel histogram of all potentially-occluded objects $H_i^{(k)}, i \in [1, N], k \in [1, M]$, where N is the number of superpixels and M is the number of potential objects, and per superpixel histogram of an occluded area $H_i', i \in [1, N]$.

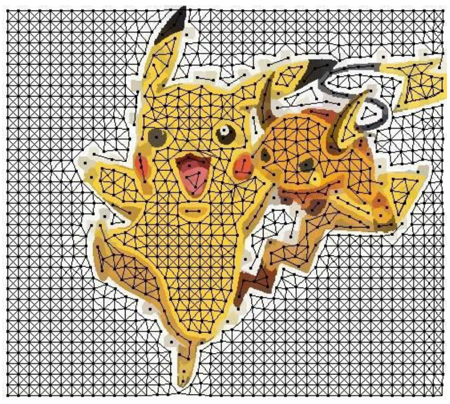
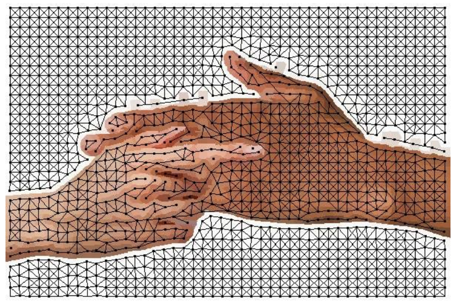
$$D_p(f_p) = \min_{i \in [1,N]} \left\| H_p' - H_i^{(f_p)} \right\|_1$$

$$V_{p,q}(f_p, f_q) = w_{p,q} \cdot 1_{f_p \neq f_q}, \qquad w_{p,q} = \infty \cdot 1_{TC}, \qquad (p, q) \in Nb$$

Data cost of assigning f_p to superpixel p is the minimum of absolute bin difference between superpixel p and any member superpixel of the object labeled f_p. Smoothness cost between two superpixel p, q with labels f_p, f_q is product of link weight $\mathrm{w}_{\mathrm{p,q}}$ and the indicator, which output 1 if $f_p \neq f_q$ and 0 otherwise. Link weight depends on the result of threshold cut on RAG.

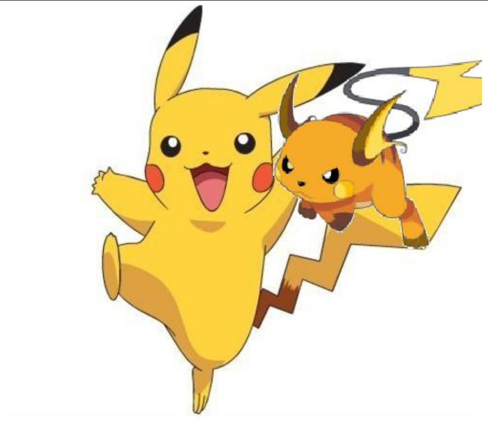
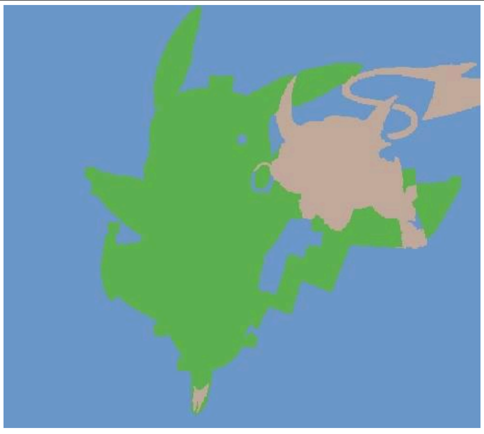
- ✓ Threshold cut:

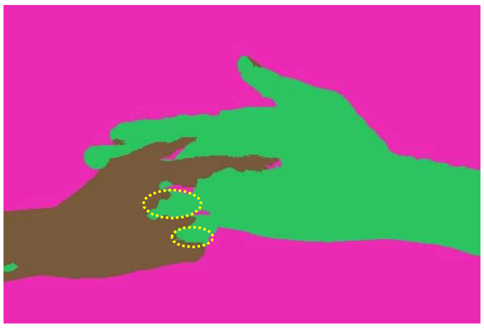
Threshold cut is a pretty simple algorithm. We first define a certain measure between two nodes, and then apply a threshold over measurement to cut the graph. In our case, differences between centroids of two adjacent superpixels are computed. If the difference between two superpixels is higher than certain threshold, cut the edge between two superpixels. It can be easily visualized as the following picture.

The main purpose of using threshold cut is to gather up all similar superpixels, and give the same cluster of superpixel strong bonds, multiple of infinity. Threshold cut and SLIC together can somehow viewed as hierarchical clustering, giving a graph with distinctive size of merged nodes. From other perspectives, the first two clustering is unsupervised learning, and the final graph cut algorithm determines to which objects an observation belongs.

- Experimental result:

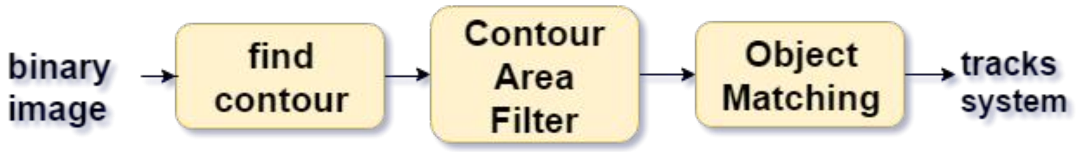





We tried to use similar objects to test the strength of our method. The upper Pikachu image contains Pikachu and upgraded Pikachu. Both of them have similar colors. The result is pretty good regardless of few glitches. The lower image with two occluded hands is used to simulate occlusion over swimmers. As we can see in the yellow dashed circle, there are huge defects over segmentation. This is because the failure of combination of threshold cut link weight (observe figure in threshold cut section, the error parts are connected to the right object). Note that the misclassification happens in shadowed area, and in reality, this may frequently happen. From our perspective, success in Pikachu image results from simple color distribution in superpixels, whereas the two-hands image depicts a 3D world, which may be much complicated due to variation of lightness. Accordingly, we still have to find a better way to deal with this problem.

3. Multiple object tracking:

   After all objects including the occluded are well-segmented, we need to keep track on each object at every time points. Our implementation can be basically separated into 3 stages, shown as follows,



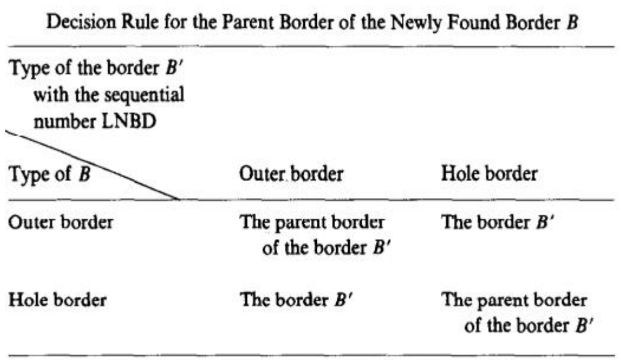
First of all, we find the contour of every objects. Then, in order to eliminate noises, we abandon those with too small contour area. Last, all detected objects go through matching process associated with tracks with objects at previous time point. Finally, we can compute and record feature of objects on all occupied tracks.

- Find contour:

  There are four steps to find the contours of a digitized binary image, before starting the program, set NBD to 1(NBD is the variable that help us construct the topology structure of contours in the image), and create variables (i, j), (i1, j1), (i2, j2), (i3, j3), (i4, j4), to store pixel coordinate values. To find the starting point of a contour, we scan through the image from top left corner, row by row, left to right.

  1. When pixel (i, j) is not zero
     a. If pixel (i, j+1) is 0, it is a outer border, increase NBD by 1, and assign (i, j-1) to (i2, j2).
     b. If pixel (i, j) is greater or equal to 1 and pixel (i, j+1) is 0, it is a hole border, increase NBD by 1, and assign (i, j+1) to (i2, j2). If pixel (i, j) is greater than 1, assign that value to LNBD.
     c. If either the condition above satisfied, then jump to step four.

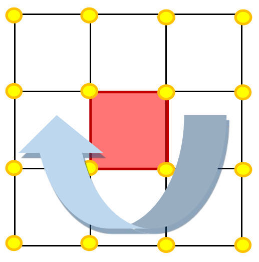
2. Depending on the types of the newly found border and the border with the sequential number LNBD (i.e., the last border met on the current row), decide the parent of the current border as shown in the table.



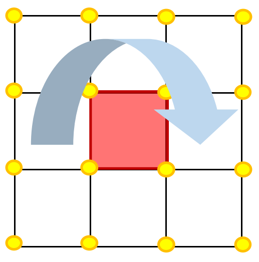
Decision Rule for the Parent Border of the Newly Found Border *B*

Type of the border *B'* with the sequential number LNBD / Type of *B*	Outer border	Hole border
Outer border	The parent border of the border *B'*	The border *B'*
Hole border	The border *B'*	The parent border of the border *B'*

3. There are five stage in step 3.
   a. Starting from (i2, j2) (centered at (i, j)), scan clockwise the neighborhood and find the nonzero pixels. If found, assign the first nonzero pixel's coordinate value to (i1, j1). If nothing is found, let pixel value of (i, j) equal to -NBD and jump to step four.



   b. Assign (i1, j1) to (i2, j2), assign (i, j) to (i3, j3).
   c. Starting from next element(from (i2, j2) in counterclockwise), centered at (i3, j3), scan counterclockwise the neighborhood and find the nonzero pixels. If found, assign the first nonzero pixel's coordinate value to (i4, j4).

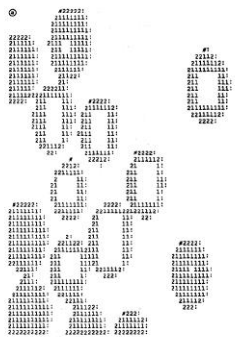


   d. If (i3, j3+1) is 0 while doing 3-3, then change pixel (i3, j3)'s value to -NBD. Otherwise, if (i3, j3+1) is not 0 while doing 3-3 and pixel value of (i3, j3) is 1, change value of (i3, j3) to NBD.
   e. If (i4, j4) is equal to (i, j) and (i3, j3) is equal to (i1, j1), this means we

are back at the beginning, thus jump to step four, otherwise, we change (i2, j2) to (i3, j3), (i3, j3) to (i4, j4) and repeat process 3-3.

4. If pixel (i, j)'s value is not equal to 1, then change LNBD's value to the absolute value of pixel (i, j), finally resume the raster scan from pixel (i, j+1), going back to step one again. The ending condition is that when raster scan counters the bottom right corner.

To extract only the outermost borders of an image(no parenting), just simply set NBD to 2 and set LNBD to 0 when scanning a new row of the image. This is a sample of an image after passing the process.



- Contour area filter:

We use a function in opencv called cvContourArea to calculate the area of a contour. The function will return how many nonzero pixels there are within the contour. The method is based on Green's formula. Green's formula uses an alternative way, double integral, to calculate a line integral. Green's formula is as follow.

$$\oint_C (Ldx + Mdy) = \iint_D (\frac{dM}{dx} - \frac{dL}{dy})dxdy$$

C is a positively simple close curve in a plane, and D is the region bounded by C. The equation is established if L and M are functions of (x, y) on an open region containing D and have continuous partial derivatives there. In our case, we can simply change the formula into below because each pixel contributes one unit area.

$$\iint_D 1dA$$

Using a for loop, when scanning through the border points row by row, the value will tell if it is the end of the area of the row(semicolon indicates the end), then we can know how many pixels are there in a single row of the contour. After calculating all the rows the contour area occupies, we can get the total area of the contour.

Another for loop runs through the contours and eliminates those contours with relatively small area. In the end we can filtered out unwanted contours.

- Object matching:

  After the stage of “contour area filter”,

  - ✓ Matching score estimate:

    There are two steps in Matching Algorithm. First, we need to estimate the matching score. At the beginning, let all occupied track’s centroids be filtered by “Kalmen Filter”, so that we can obtain predicted centroids of all occupied tracks. Then, we calculate the distances between every ND’s centroid and predicted centroids. Finally, the smaller one between the distance and truncated value is the estimated matching score of that pair of ND and track.

  - ✓ Munkres algorithm:

    At this point, score of each track-ND pair is known, so we can apply it to “Munkres Algorithm”, which is the second step in Matching Algorithm. First, we align those scores in a form of M * N matrix, when there are M occupied tracks and N NDs. That is, element locates in 1st row and 1st column is the score of track1-ND1, element locates in 1st row and 2nd column is the score of track1-ND2, and element locates in Mth row and Nth column is the score of track(M)-ND(N), etc. Use this Matrix as input, and Munkres output a permutation matrix of occupied track-ND pair with lowest cost. For example (the picture below), there are 4 occupied tracks and 4 NDs, so we have a 4x4 matrix as Munkres Algorithm’s input, and it will output a sparse matrix with only one nonzero value for each row and each column, while all the other elements in the output matrix are zero. That’s say track1-ND2, track2-ND1, track3-ND4, and track4-ND3, these 4 elements are nonzero, so that these 4 pairs are our final results, which means ND2 belongs to track1, ND1 belongs to track2, ND4 belongs to track3 and ND3 belongs to track4.

$$\begin{array}{c c} & \begin{array}{cccc} \text{ND1} & \text{ND2} & \text{ND3} & \text{ND4} \end{array} \\ \begin{array}{c} \text{Track1} \\ \text{Track2} \\ \text{Track3} \\ \text{Track4} \end{array} & \left[\begin{array}{cccc} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{array}\right] \end{array}$$

    There are typically 4 steps within Mukres Algorithm. The first two steps are executed once, while steps 3 and 4 are repeated until an optimal assignment is found. First of all, we find the smallest element for each row, and subtract it from each element in that row. Second, we also find the smallest element for each column, and subtract it from each element in that column. Third, using a minimum number of horizontal and vertical lines to cover all zeros in the resulting matrix. If n lines are required, an optimal assignment exists among the zeros. The algorithm stops. Else, we continue with the last step if less than n lines are required. Lastly, to find the smallest element (call it k) that is uncovered by a line in step 3, subtracting k from all uncovered elements, and add k to all elements that are covered twice.

✓ Mathematical expression of Munkres Algorithm:

The mathematical definition of Munkres Algorithm can be represented by following equations:

$\{c_{ij}\}_{N\times N}$ – cost matrix, where $\boldsymbol{c_{ij}}$ – cost of track ***i*** that matches ND ***j***.

$\{x_{ij}\}_{N\times N}$ – resulting binary matrix, where $\boldsymbol{x_{ij}}$ = 1 if and only if ***i***th track is assigned to ***j***th ND.

$\sum_{j=1}^{N} x_{ij} = 1, \quad \forall i \epsilon \overline{1,N}$ – one track to one ND assignment.

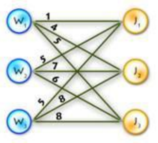
$\sum_{i=1}^{N} x_{ij} = 1, \quad \forall j \epsilon \overline{1,N}$ – one ND to one track assignment.

$\sum_{i=1}^{N}\sum_{j=1}^{N} c_{ij}x_{ij} \rightarrow min$ – total cost function.

We can also transform this problem into graph theory. To view tracks and workers as if they were a bipartite graph, where each edge between the ***i***th track and ***j***th job has weight of $\boldsymbol{c_{ij}}$. Then our aim is to find minimum-cost matching in the graph (the matching will consists of **N** edges, because our bipartite graph is complete).

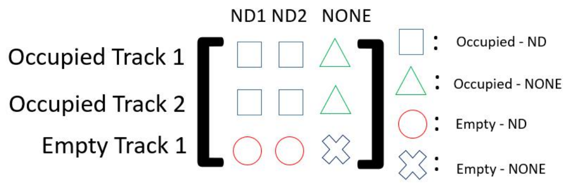
The picture below is an example for more clarity:

$$\begin{pmatrix} 1 & 4 & 5 \\ 5 & 7 & 6 \\ 5 & 8 & 8 \end{pmatrix} \Leftrightarrow$$



✓ Implementation:

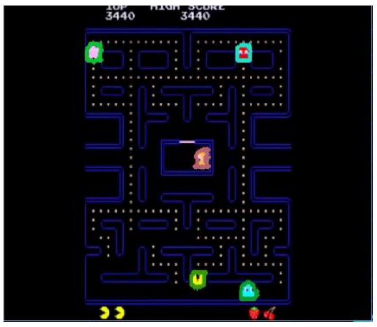
As for our implementation, we first assign all occupied tracks and then all empty tracks in row, while assign all NDs in column. Moreover, to make the matrix a square matrix, we make up the deficit in column with "NONEs". For instance, if we have 2 occupied tracks, 1 empty track, and 2 NDs, then row 1~2 represent occupied tracks, row 3 represents empty tracks, column 1~2 represent NDs and column 3 represents NONE. For all pairs (empty, ND), (empty, NONE), (occupied, ND), we assign them truncate values. Finally, applying this matrix to Munkres Algorithm, and we can obtain a permutation matrix. If any pair corresponding to the permutation matrix has score equal to truncate value, then we delete that track and create a new track for that ND.

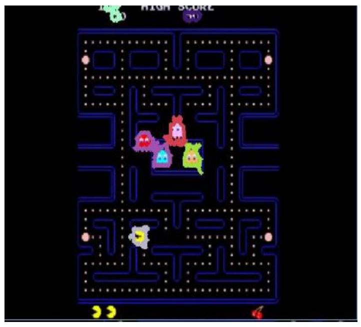


After the whole process of Matching Algorithm, which contains two steps, Matching Score Estimate and Munkres Algorithm, we can classify the result into three different situations. First, ND X matches track X. Second, track X has no corresponding detection, which means track X has to be deleted. Third, ND X has no corresponding track, which means a new track needs to be created by using an empty track.

Until now, the entire procedure of Multiple Object Tracking is finished, and we are able to track each object individually.

- ✓ Experimental result:





As you can see in the above pictures, which are the experiment result for a video game clip, we can successfully track multiple objects with different tracks shown by different colors. However, it remains a problem that needs to be resolved. That is, an object may sometimes changes its track's color that it should not do. The main reason is that the dependency we use to determine tracks is the centroid of its contour. As a result, when the location of the centroid of a certain object is way too far to the last time frame it was, that object would be regarded as two different objects in two time frames, but actually it is the same object, so it causes incorrect tracking. Furthermore, the method we calculate centroids is influenced by contours. Hence, if we failed to do well in our previous step, background model, it may easily induced to noises in contours and further cause error in calculation of centroids. There is another point worth mentioned, we observe that an object may disappear in somewhere which is not border, and at the same time detects some other objects around it. This is the situation that this disappeared object has happen occlusion with its nearby objects.

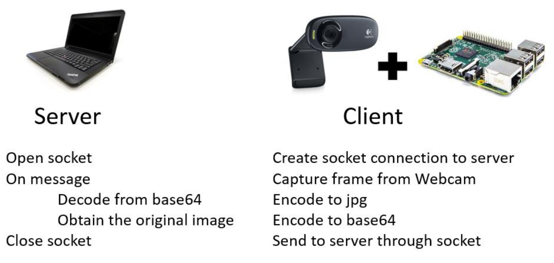
4. Hardware - Raspberry Pi 2 + Webcam:

- Client:

  On Raspberry Pi 2 with Webcam, we run a client program written in Python. First, connecting with server using Python's Websocket module. As soon as Websocket is established and connection between server and client succeed, Webcam starts capturing each frame one after another. Then, for each captured frame, several encoding process is required. We first encode the original frame into .jpg image, then turning it into the form of Numpy array, and at last encode into Base64 format. Finally, we can send this Base64 code which represents the captured frame to the server through Websocket.

- Server:

  On our host computer, we run a server program written in Python as well. In the first place, taking advantage of Python's Tornado Server module, we can easily build up a server and open a Websocket connection through Tornado Websocket Handler. Whenever a new client connects to this server, it begins to receive Base64 codes sent from that client, but it doesn't cause any negative effect to previous connected clients, each client works independently and fine with this central server. After receiving Base64 codes, obviously, we need to decode them to recover to original frames. As a result, doing the inverse way that clients do. First, the message is decoded into Numpy array, and further decoded into original image. From now on, we are able to exploit those images captured form the corners of the swimming pool and do the following identification process.



```
Open socket
On message
        Decode from base64
        Obtain the original image
Close socket
```

```
Create socket connection to server
Capture frame from Webcam
Encode to jpg
Encode to base64
Send to server through socket
```