ACS61012 Machine Vision

Saba Firdaus Ansaria
Reg No: 210110201

March 2022

Contents

1 Task 1 Introduction to machine vision **1**
1.1 Part I . . . 1
1.2 Part II . . . 3
1.2.1 Sobel . . . 4
1.2.2 Canny Edge . . . 4
1.2.3 Prewitt . . . 5

2 Task 2: Optical flow estimation algorithm **6**
2.1 Part I Gingerbread Man . . . 6
2.2 Part II Red square Image . . . 6

3 Task 3: Automatic detection of moving objects in a sequence of video frames **8**
3.1 Part I: with the frame differencing approach . . . 8
3.2 Part II: with the Gaussian mixture approach . . . 10

4 Task 4: Treasure hunting **14**
4.1 Algorithm I (easy, medium scenario) . . . 14
4.2 Algorithm II (hard scenario) . . . 17
4.3 Discussion . . . 23

5 Task 5: Siamese CNN, YOLO CNN, Capsule CNN **23**
5.1 CNN . . . 23
5.2 Siamese CNN . . . 25
5.2.1 Contrastive Loss . . . 25
5.2.2 Triplet Loss . . . 26
5.3 YOLO CNN . . . 26

5.4 Capsule CNN . . . 27
5.5 Comparison and Application . . . 28
5.5.1 Model size . . . 28
5.5.2 Representation Learning . . . 29
5.5.3 K-shot Learning . . . 29
5.5.4 Application . . . 29

6 Appendix **30**
6.1 Code for problem 2 . . . 30
6.2 Problem 4 code . . . 34
6.2.1 Problem 4 Easy Medium Scenario . . . 34
6.2.2 Task 4 Hard Scenario . . . 36

List of Figures

1 Image histogram different formats and channels . . . 1
2 Image histogram adding different brightness across different formats and channels . . . 2
3 Image histogram reducing different brightness across different formats and channels . . . 3
4 Edges with sobel filter and different thresholds . . . 4
5 Edges with canny filter and different thresholds . . . 5
6 Edges with prewitt filter and different thresholds . . . 5
7 Corner points on Gingerbread Man . . . 6
8 Optical flow on Gingerbread Man (full image) . . . 7
9 Optical flow between Gingerbread Man frames . . . 7
10 Last Frame bounding box and optical flow visualisation with ground truth and predicted point value.The green arrows represent the flow vectors. . . . 8
11 Prediction error over frames. RMSE over the frames is 0.5551 . . . 8
12 Change detection with frame difference and varying thresholds at the same time frame. . . . 9
13 Change detection with Gaussian mixture model and varying number of Gaussian and prior. . . . 11
14 Change detection with Gaussian mixture model and varying number of Gaussian and prior. . . . 12
15 Change detection with Gaussian mixture model and varying number of Gaussian and prior. . . . 13
16 Easy Treasure Hunt (The treasure is in green box after 10th box) . . . 14
17 Medium Treasure Hunt (The treasure is in green box next to 7th box) . . . 15

18 Hard Treasure Hunt path discovery (treasure green leaf and yellow sun) . . 18
19 A general CNN architecture comprising several layers 24
20 A general CNN siamese architecture comprising triplet/contrastive loss . . 25
21 Typical GoogleNet block (blog.paperspace.com) base for YOLO models . . 26
22 A representation of YOLO perspective of several glances/instances of a image 27
23 A representation of CapsuleCNN as shown in [1]. 28

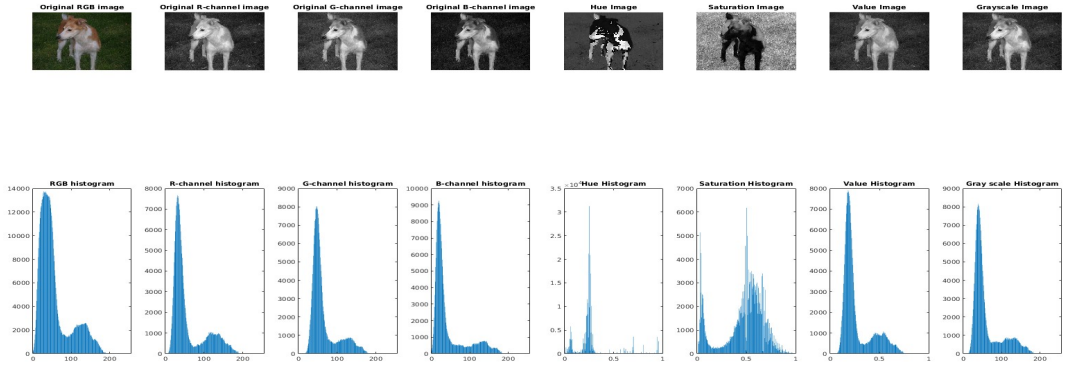
List of Algorithms

1 Treasure hunt Algorithm main structure.(approach 1) 15
2 Treasure hunt FindArrow function 16
3 Treasure hunt RedArrow function 16
4 Treasure hunt nextObjectFinder function (approach 1) 16
5 Treasure hunt Algorithm main structure.(Approach 2) 19
6 Treasure hunt nextObjectFinder2 function (approach 2) 20

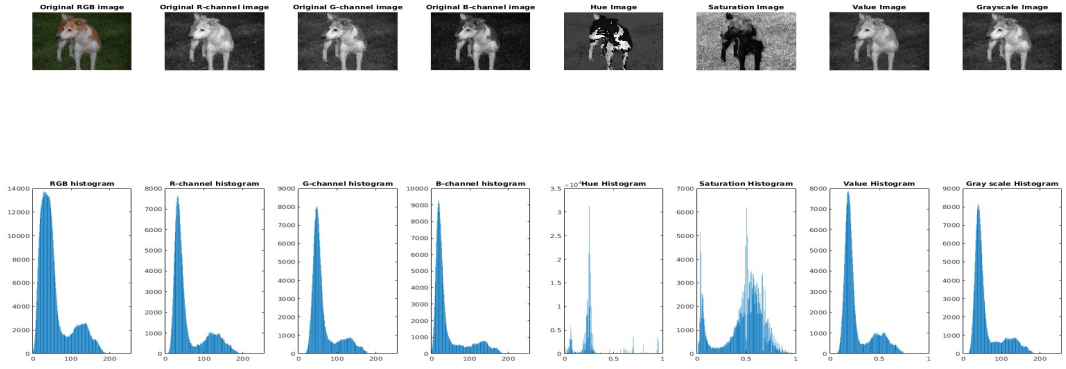
1 Task 1 Introduction to machine vision

1.1 Part I

Image Histogram is the graphical representation of a digital image representing the frequency of pixels as a function of their intensity. A graph is a plot with its horizontal axis representing the intensity variation and the vertical axis representing the frequency of pixels for that specific intensity value. Figure 1,2,3 shows different image transformations and their corresponding histograms.



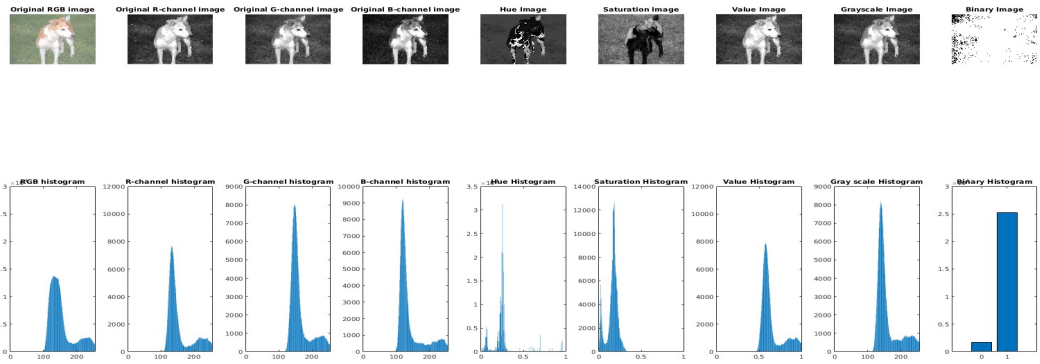
(a) Histogram with the image



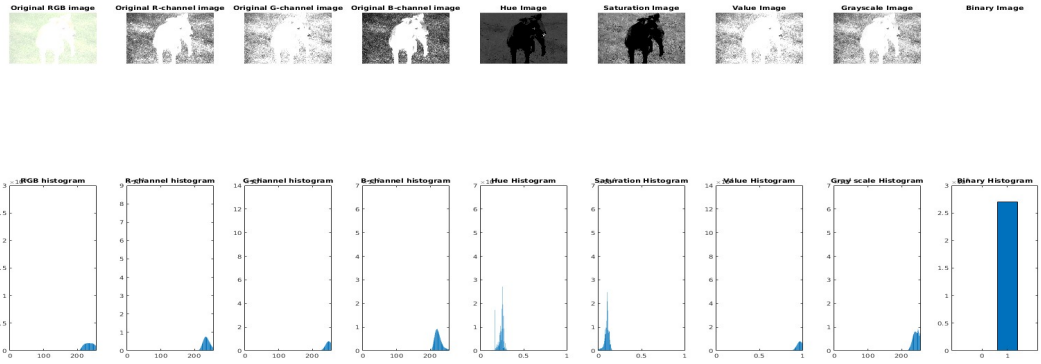
(b) Histogram with the flipped image

Figure 1: Image histogram different formats and channels

The horizontal axis extends on both the left and right sides. The left side denotes the darker part, whereas the right side denotes the lighter part of the image, and the central portion represents the mid-intensity values of the image. From simple observation, it can be seen that the histograms of the darker images in Figure 1 and reduced brightness images in Figure 2 have the left side of the horizontal axis high. On the contrary, in Figure 2, the image brightness is increased, and the right side of the horizontal axis is



(a) Histogram with added 100 brightness



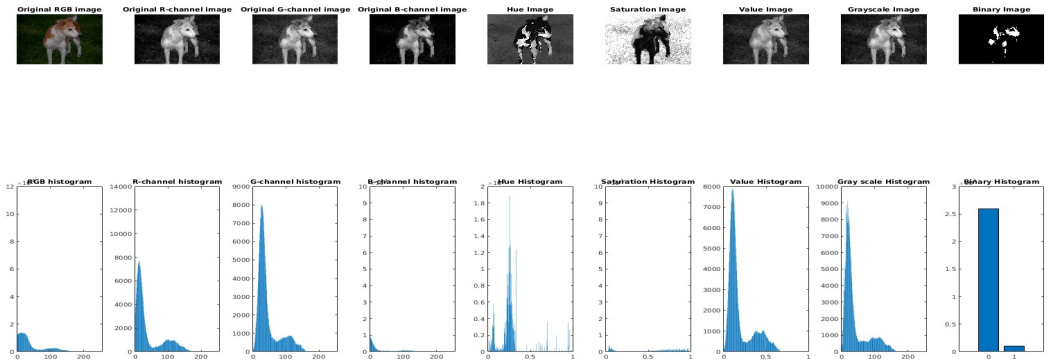
(b) Histogram with added 200 brightness

Figure 2: Image histogram adding different brightness across different formats and channels

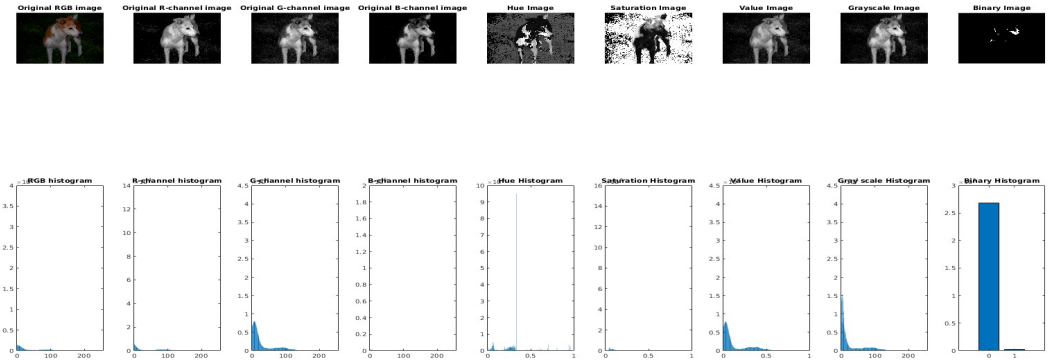
high. Hence for a general image, if the frequency of the bars lying on the left side of the horizontal axis is high, then the image is darker, whereas if the frequency of the bar is high on the right side of the horizontal axis, then the image is lighter.

Image histograms are used in image processing, image analysis, equalizing an image, and thresholding. By just observing the image's histogram, one can predict the information related to the image. The histogram can regulate the brightness of the image and adjust the contrast of an image. If the histogram has the pixel counts evenly covered over a broader range, this denotes an image with good contrast, whereas the limited pixel count to a smaller range denotes a low contrast. This characteristic is clearly visible between the normal image in Figure 1 and the saturated image in Figure 1.

These characteristics and observations help make image corrections as well. For example, the pixel frequency of Figure 3 is high on the far-right side of the histogram(this implies that the image is much lighter). By using this information, the image can be



(a) Histogram with reduced 20 brightness



(b) Histogram with reduced 40 brightness

Figure 3: Image histogram reducing different brightness across different formats and channels

improved by shifting the values towards the available central range of intensity so that the pixel values are evenly spread.

1.2 Part II

Edge Detection is a fundamental tool in machine vision that uses image processing techniques specifically in the feature extraction and feature detection region. It consists of several mathematical methods for identifying edges in a digital image where there is a sharp change in the brightness of the image. The motivation for this change is to preserve these important events. The discontinuity in the image's brightness corresponds to the discontinuities in the depth, surface orientation, properties of the material, and illumination variations.

1.2.1 Sobel

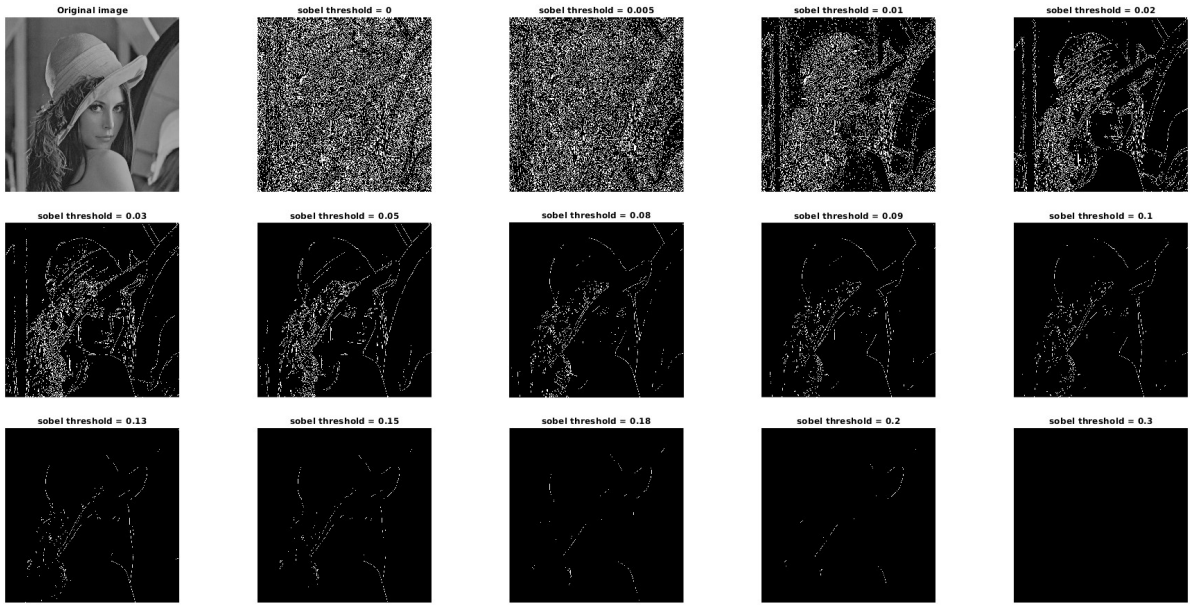


Figure 4: Edges with sobel filter and different thresholds

Sobel Edge Detector is one of the most acceptable edge detection algorithms in image processing techniques, which is a 3×3 discrete differentiation operator that computes the gradient of the function of the intensity of the image. Further, it produces a result which is either the respective gradient vector or the norm of the given vector.

This filter produces an image which focuses on the edges. The Sobel operator requires fewer computations as it revolves around the fundamentals of image convolving consisting of small, detachable integer-valued filters in both the horizontal and vertical direction. The optimal sobel threshold is between 0.03 and 0.09. If the threshold reaches near 0, the edges become noisy and if the threshold goes past 0.2, the edges become undetectable. The demonstration with various threshold values for sobel is shown in Figure 4.

1.2.2 Canny Edge

Canny Edge Detector involves an algorithm based on multi-stage, detecting the edges in different ranges of the images. To calculate the intensity of the gradients, it uses a filter based on the Gaussian derivative. This derivative of the Gaussian reduces the noise in the image. Furthermore, in the gradient magnitude, the non-maximum pixels are removed, resulting in thinning of the potential edges down to 1-pixel curves. At last, the use of hysteresis thresholding on the gradient magnitude either terminates or keeps the pixels of the edge. The optimal canny edge threshold is between 0.05 and 0.5. If the threshold reaches near 0, the edges become noisy and if the threshold goes past 0.9, the edges become undetectable. The demonstration with various threshold values

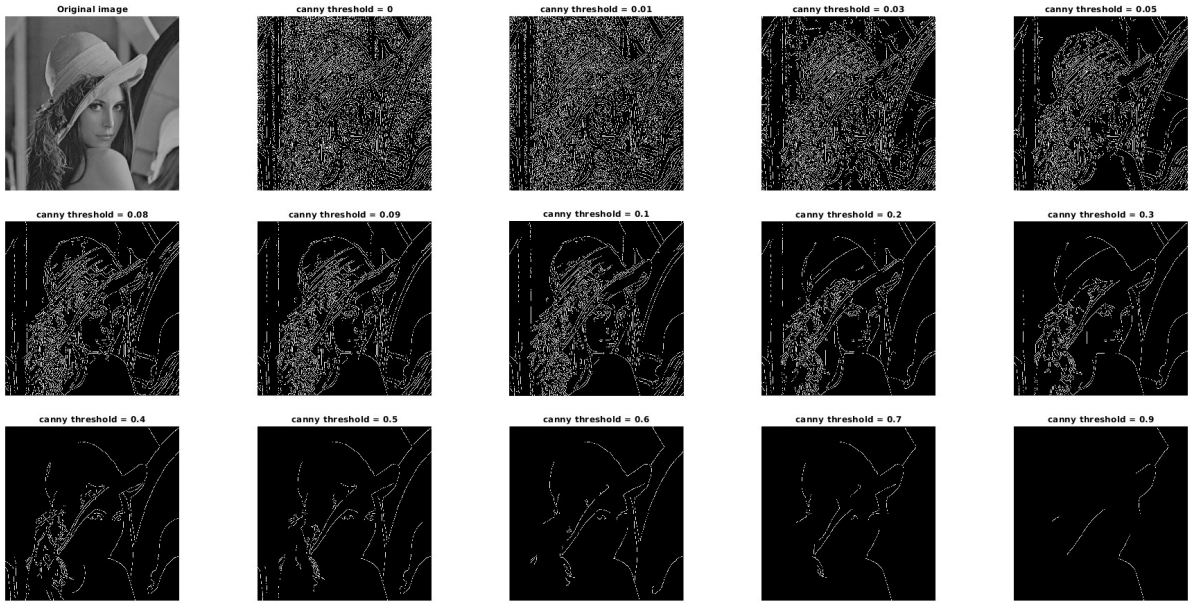


Figure 5: Edges with canny filter and different thresholds

for canny operation is shown in Figure 5. Canny edge operator has better range than sobel operation.

1.2.3 Prewitt

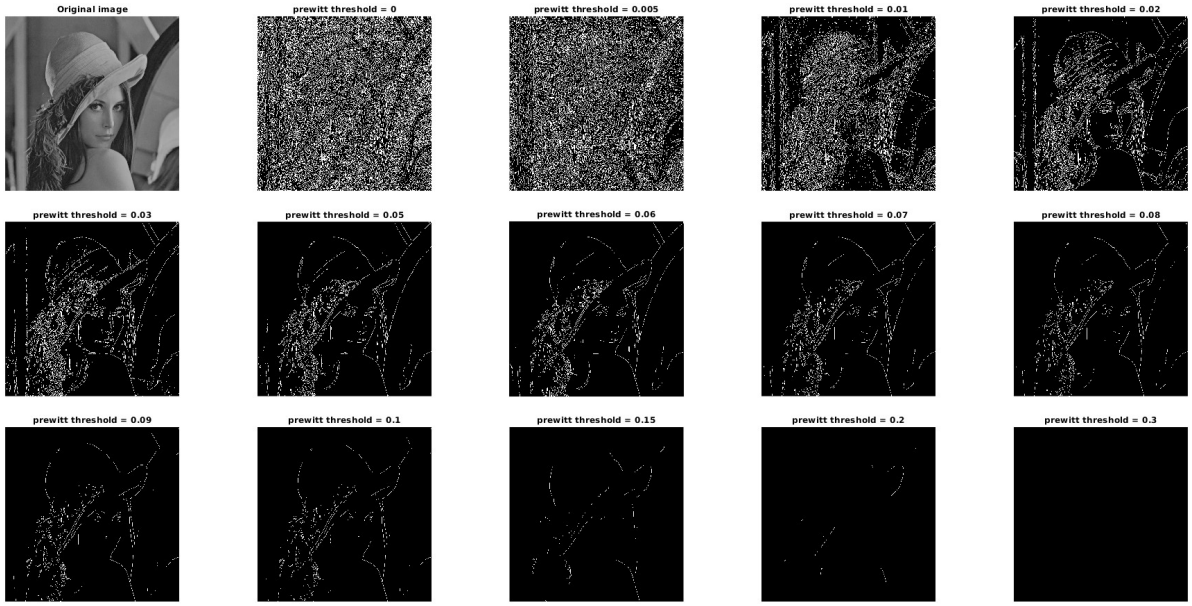


Figure 6: Edges with prewitt filter and different thresholds

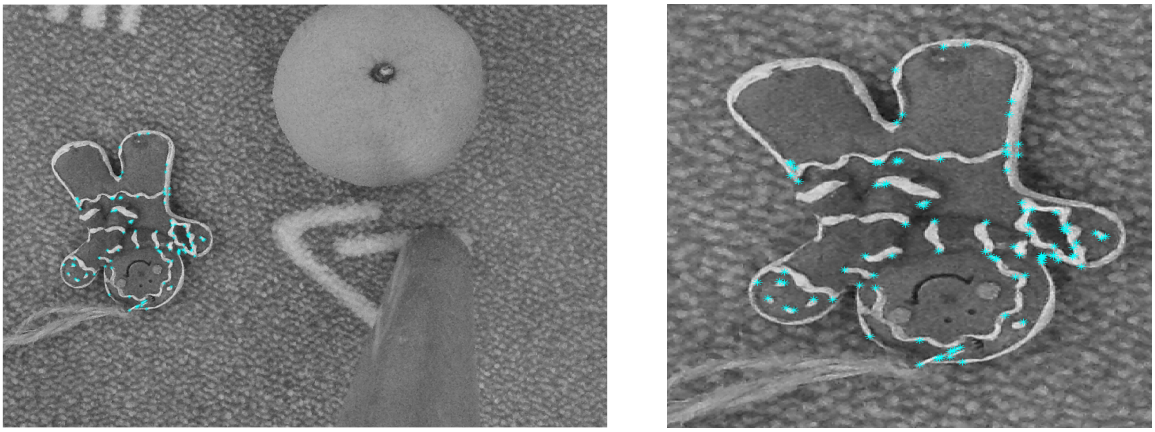
Prewitt Operator is a discrete differentiation operator which calculates the approximate value of the gradient at each point of the image intensity function. It detects the horizontal and vertical edges of the image, and it represents as gradient vector or norm of

this vector at each point of the image. Generally, it produces a simple approximation for the gradient, specifically for the high-frequency variations in the image. At each point in the image, it calculates the image intensity gradient. This point provides the direction of the most significant possible gain from light to dark region and the rate of change. The optimal prewitt threshold is between 0.09 and 0.06. If the threshold reaches near 0, the edges become noisy and if the threshold goes past 0.2, the edges become undetectable. The demonstration with various threshold values for prewitt operation is shown in Figure 6.The value of the magnitude of the edge is more accurate and simpler to comprehend than the direction at the point in the image. Prewitt has the smallest threshold range between these three operators.

2 Task 2: Optical flow estimation algorithm

2.1 Part I Gingerbread Man

The corner points on Gingerbread Man and its magnified version is shown in Figure 7. The optical flow between two frames of Gingerbread Man is shown in Figure 8 and the magnified image is shown in Figure 9.



(a) Hundred corner points (b) Magnified image hundred corner points

Figure 7: Corner points on Gingerbread Man

The maximum corner points that can be drawn using *corners* is 200. In this case, hundred corner points have been drawn. The resulting images are magnified for better visualisation.

2.2 Part II Red square Image

The pixel tracking algorithm using optical flow and bounding box has been performed as per the instruction given in the course tutorial. The flow vectors are visualised and the

Figure 8: Optical flow on Gingerbread Man (full image)

(a) Optical flow vectors.

(b) Optical flow vectors on Gingerbreadman (magnified)

Figure 9: Optical flow between Gingerbread Man frames

ground truth is compared with the predicted flow cor-ordinates. At each frame the loss is stored and it has been plotted in Figure 11.

The last frame of the red square video is shown in Figure 10 and the plotting for prediction errors over frames is shown in Figure 11. The mean squared error over all the frames is `0.5551`. The arrows over the red box in Figure 10 signifies the optical flow vectors.

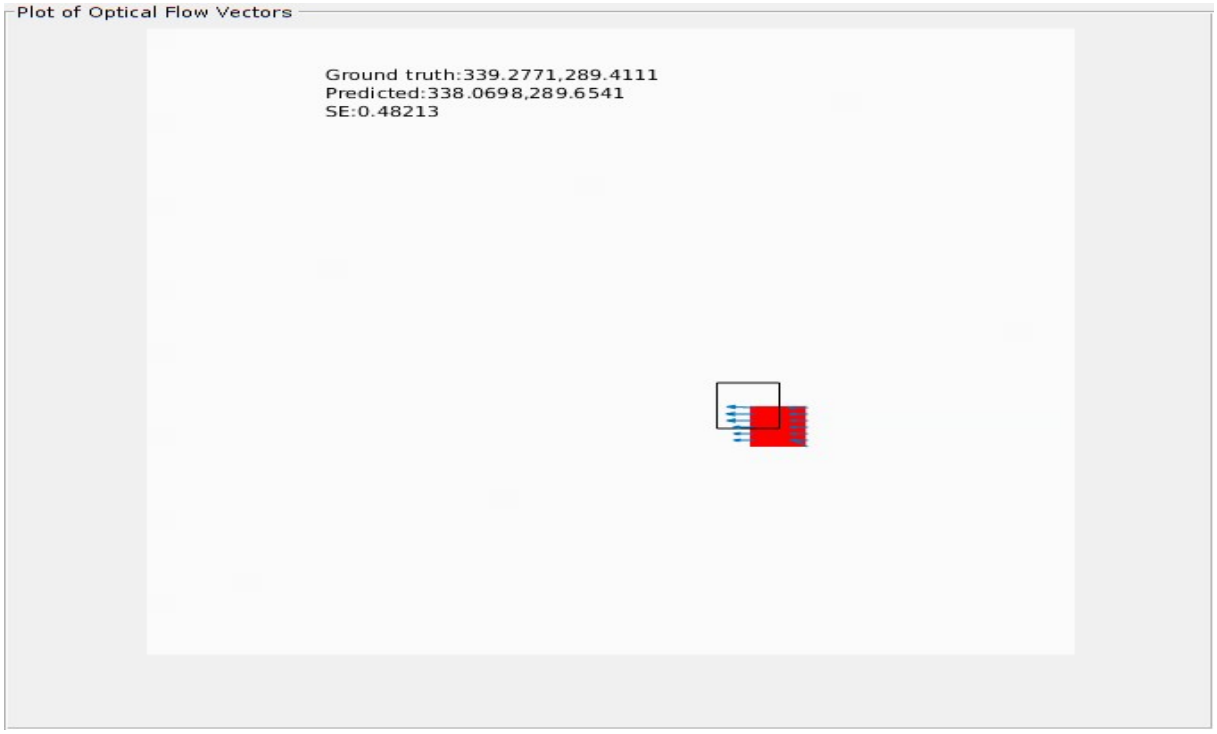


Figure 10: Last Frame bounding box and optical flow visualisation with ground truth and predicted point value.The green arrows represent the flow vectors.

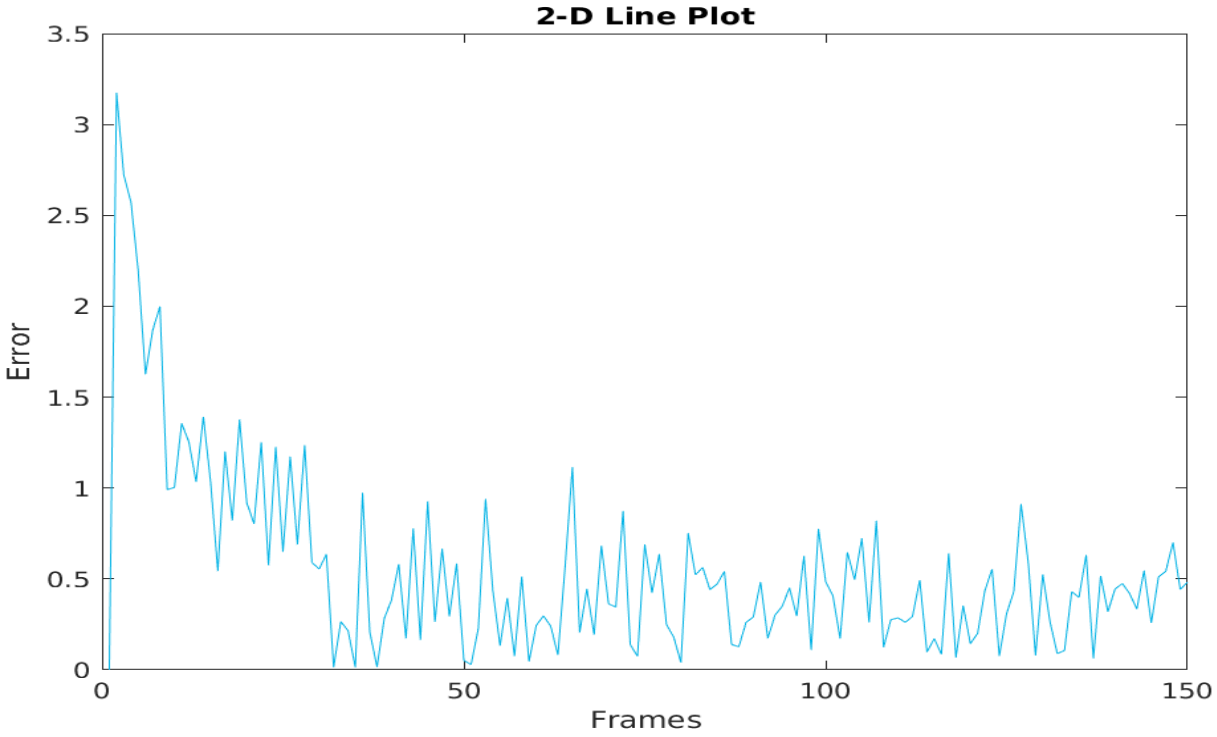


Figure 11: Prediction error over frames. RMSE over the frames is 0.5551

3 Task 3: Automatic detection of moving objects in a sequence of video frames

3.1 Part I: with the frame differencing approach

In the frame differencing approach, a video with n=1,.2,...N number of frames can be used by simply taking the previous frame as a background model and considering the

(a) Original Image.

(b) Frame difference (threshold =1).

(c) Frame difference (threshold =25).

(d) Frame difference (threshold =50).

(e) Frame difference (threshold =100).

Figure 12: Change detection with frame difference and varying thresholds at the same time frame.

pixel differences as a measure of change between two consecutive or two different time frames. These frames are firstly converted to grayscale, and then the pixel difference is calculated. However, this approach has some obvious flaws

1. As the change is only detected by pixel difference, the method is vulnerable to lighting difference. Because the lighting conditions can change a pixel intensity.

2. Similarly, this method is very much vulnerable to camera jitters or any platform-related noise. However, this can be managed by setting proper threshold values.

3. The difference threshold value is crucial, and it is explored here with different threshold values.

4. The lower the threshold is, the higher the noise and false positive. The higher the threshold, the lower the noise and false positive. However, if the threshold is too high, the change becomes undetectable eventually.

5. Different change detection events have been demonstrated in Figure 12. It is very clear that with increasing threshold the noise is reducing in Figure 12.b and 12.c.

6. However, it is evident that with increasing threshold, the object representation and the background became non-existent, as shown in 12.d and 12.e. All, the frames of different thresholds have been measures in the same time frame.

3.2 Part II: with the Gaussian mixture approach

One major drawback of the frame differencing approach is that it assumes pixels will remain the same between two still instances. This is not a reasonable assumption because pixel values change on lighting conditions, recording devices etc. The GMM model trains Gaussian distribution with prior image frame data assuming that the pixel values may change. The intensity distribution is modelled as a mixture of Gaussian, and the previous frames are used as prior. This approach is implemented in MATLAB, in the Computer Vision System Toolbox. Just to compare fairly, all the parameter variations are displayed with the exact same frame in the video. The number of prior frames and the number of Gaussian have varied to understand the GMM foreground modelling. The following findings are given below.

1. The number of Gaussian has a significant role in background modelling. When the number of Gaussian is increased, it produces a better and more descriptive worldview model with prior data. This is evident in Figure 15. In Figure 15 the number of Gaussian is increased to $5, 10, 15, 80$, and it can be clearly seen that more details have been represented with increased Gaussian components. The better representation happens when the number of prior is also increased significantly with the number of Gaussian. Otherwise, the modelling improvement does not occur significantly (Figure 14).

2. In Figure 14, the Gaussian is increased while keeping the prior data the same (100 frames). It can be clearly observed that no visible change has happened in the background model. This implies that the number of prior frames is as crucial as the number of Gaussian.

3. Similarly, behaviour is noticed in Figure 13 where the prior frames remain the same while the number of Gaussian is increased. There is no visible change in the background modelling.

(a) Original Image.

(b) GMM movement change detection(prior frames=10,Gaussian =2).

(c) GMM movement change detection(prior frames=10,Gaussian =5).

(d) GMM movement change detection(prior frames=10,Gaussian =20).

(e) GMM movement change detection(prior frames=10,Gaussian =30).

Figure 13: Change detection with Gaussian mixture model and varying number of Gaussian and prior.

4. Overall, the observation gives the inference that the background modelling is also subjected to the complexity of the background model. If the background model is not static and complex, an increasing number of Gaussian and prior frames will help. However, a large number of Gaussian or prior context data does not show many improvements in the case of static background modelling.

(a) Original Image.

(b) GMM movement change detection(prior frames=100,Gaussian =5).

(c) GMM movement change detection(prior frames=100,Gaussian =20).

(d) GMM movement change detection(prior frames=100,Gaussian =100).

Figure 14: Change detection with Gaussian mixture model and varying number of Gaussian and prior.

(a) Original Image.

(b) GMM movement change detection(prior frames=10,Gaussian =5).

(c) GMM movement change detection(prior frames=20,Gaussian =10).

(d) GMM movement change detection(prior frames=40,Gaussian =15).

(e) GMM movement change detection(prior frames=100,Gaussian =80).

Figure 15: Change detection with Gaussian mixture model and varying number of Gaussian and prior.

4 Task 4: Treasure hunting

The treasure hunting task requires solving three scenarios. The main challenges are finding the arrow objects and the nearest arrow objects that align with the direction of the current arrow objects. Two different approaches have been proposed for finding the nearest object. The main algorithms and the supporting function algorithms are presented in Algorithms 1-6. In the first approach, which is a very easy euclidean distance-based approach, I solved the easy scenario and medium scenario shown in Section 4.1 and Figures 16, 17. With the second approach, where I found the euclidean distance as well as the straight-line regression passing from centroid and a yellow pixel is shown in Section 4.2 and Figure 18.

4.1 Algorithm I (easy, medium scenario)

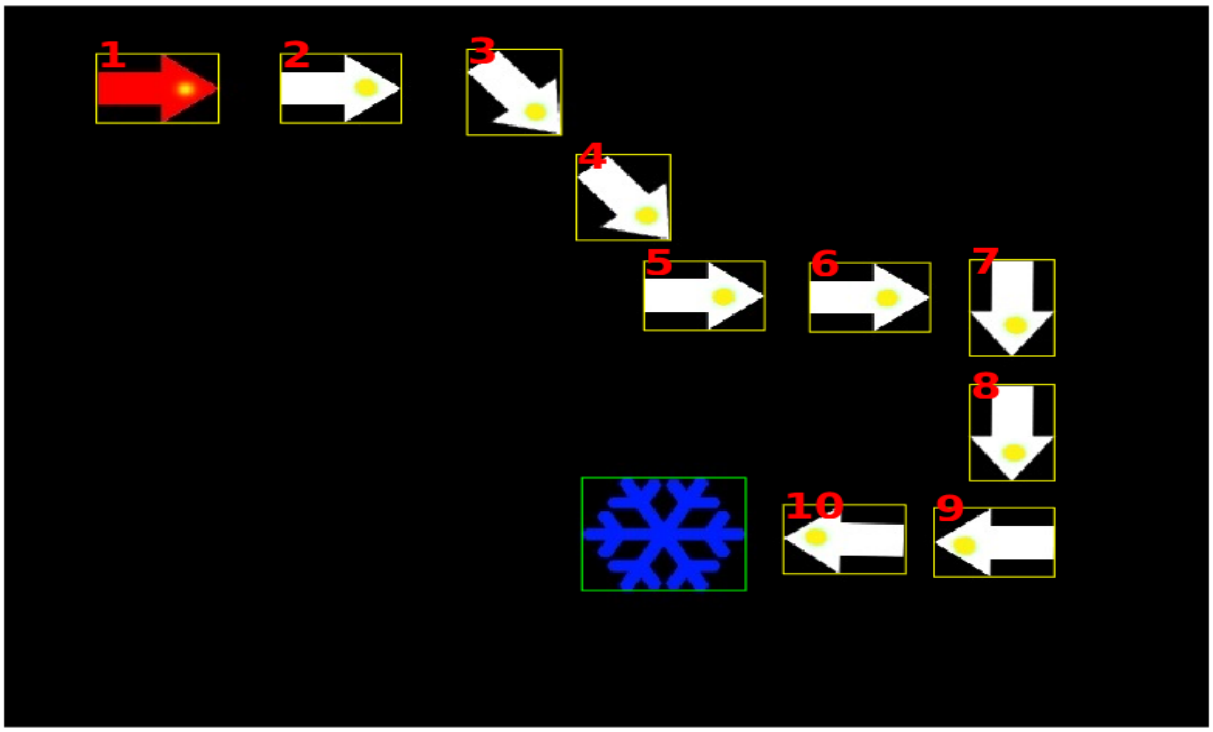


Figure 16: Easy Treasure Hunt (The treasure is in green box after 10th box)

The first treasure finding approach Algorithm I is employed for the easy and medium problem. For the Algorithm I, the main idea is given below

1. The main program flow and logical steps are shown in Algorithm 1.
2. The arrows are distinguised using the *Area* property of the bounding box discussed in Algorithm 2.
3. The next object traversal is done by taking distance between the centroids of the neighbouring objects (exclusing the already traversed one) and taking the minimum, discussed in Algorithm 4.

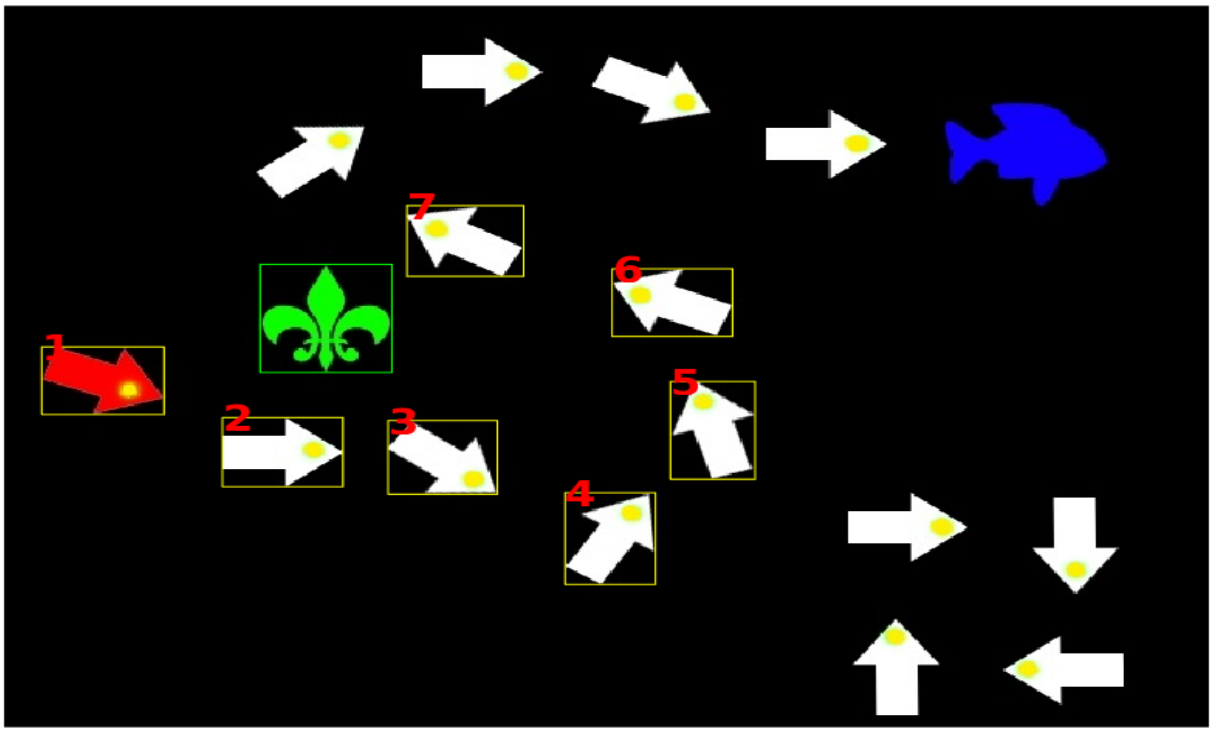


Figure 17: Medium Treasure Hunt (The treasure is in green box next to 7th box)

4. The corresponding function code snippets are shown.

Figure 16, 17 show the path discovery in easy and medium treasure hunt problem. The main algorithm steps are given in Algorithm 1. The function algorithm for finding arrow object is shown in Algorithm 2 and it is based on area of the binding box.

Algorithm 1 Treasure hunt Algorithm main structure.(approach 1)

procedure TREASURE(*image*)
    Initialise system parameters ▷ Read all the input values and initialise the params.
    im ← *read*(*image*) ▷ Read the image.
    binImage ← *binarise*(*im*) ▷ Binarisation with threshold (0.08).
    component ← *componentLabel*(*binImage*) ▷ Extract connected components.
    properties ← *regionProperty*(*component*) ▷ Extracting different comp properties.
    Draw bounding boxes for visualisation.
    arrowIds ← **FindArrow**(*properties*, *components*) ▷ Gets the component ID of arrows.
    redArrowId ← **RedArrow**(*im*, *arrowIds*, *properties*) ▷ Gets the red arrow component.
    currentId ← *redArrowId* ▷ Initial state or starting state.
    path ← *startId*
    while reach the treasure objectID **do**
        currentId ← **nextObjectFinder**(*im*, *currentId*, *properties*, *path*) ▷ Find neighbour object
        path ← *currentId* ▷ Assign the new destination in the path.
    End Loop
    Visualise the path with *path* and *im* ▷ Final Visualisation

Algorithm 2 Treasure hunt FindArrow function

```
1: procedure FindArrow(properties, components)
2:     Initialise system parameters          ▷ Read all the input values and initialise the params.
3:     areaThreshold ← 1600                  ▷ Arrow box area theshold
4:     arrowIds ← initialise                 ▷ Initialise the arrow id list.
5:     for id=1,2,..number of objects do     ▷ Run loop for all the objects.
6:         area ← properties(id).Area        ▷ Get the area of the object.
7:         if area < areaThreshold then
8:             arrowIds ← id                 ▷ Assign the Id into arrowIds
9:     End Loop
10:    return arrowIds
```

Algorithm 3 Treasure hunt RedArrow function

```
1: procedure RedArrow(im, arrowIds, properties)
2:     Initialise system parameters          ▷ Read all the input values and initialise the params.
3:     for id=1,2,..number of arrows do      ▷ Run loop for all the objects.
4:         objId ← arrowIds(id)              ▷ Get the particular object
5:         centroid ← properties(id).Centroid  ▷ Get the centroid of the object.
6:         color ← im(centroid)
7:         if color == RED then
8:             redArrowId ← objId
9:             BREAK Loop                    ▷ Assign the Id into arrowIds
10:    End Loop
11:    return redArrowIds
```

Algorithm 4 Treasure hunt nextObjectFinder function (approach 1)

```
1: procedure nextObjectFinder(im, currentId, properties, path)
2:     Initialise system parameters          ▷ Read all the input values and initialise the params.
3:     currentCentroid ← properties(currentId).Centrooid  ▷ Get the centroid of the current
   arrow object
4:     minDistance ← 999999                  ▷ Set a big value to the initial minimum neighbour distance.
5:     for objId=1,2,..number of total objects do  ▷ Run loop for all the objects.
6:         if objId not in path then         ▷ If the current object is not already in the path.
7:             tempCentre ← properties(currentId).Centrooid  ▷ Get the centroid of the
   neighbour
8:             Calculate centroid distance between the current object and neighbour
9:             distance ← DISTANCE(currentCentroid, tempCentre)  ▷ euclidean/manhattan
   distance
10:            if minDistance > distance then
11:                minDistance ← distance
12:                currentId ← objId         ▷ Assign the current nearest objId as the next neighbour
13:    End Loop
14:    return currentId
```

The equivalent code for Algorithm 4 is given below

```
function cur_object = nextObjectFinder(im, cur_object, props, n_objects,arrow_ind, path)
  current_centroid = props(cur_object).Centroid;
  min_centroid_distance = 99999;
  for object_id = 1 : n_objects
      if ismember(object_id,path) ~= true
        temp_centroid = props(object_id).Centroid;
        centroid_distance = norm(round(current_centroid) - round(temp_centroid));
        if min_centroid_distance > centroid_distance
            min_centroid_distance = centroid_distance;
            cur_object = object_id;
        end
      end
   end
end
```

The equivalent code snippet for Algorithm 2 is given below

```
    function arrow_ind = arrow_finder(props,n_objects)
    arrow_ind = [];
    for object_id = 1 : n_objects
        area = props(object_id).Area;
        centrod = props(object_id).Centroid;
        pixellist = props(object_id).PixelList;
        pixelValues = props(object_id).PixelValues;
        if area < 1600
            arrow_ind = [arrow_ind ; object_id];
    end
end
```

4.2 Algorithm II (hard scenario)

The hard scenario of the treasure finding problem is solved using the proposed Algorithm II. The main difference between the easy, medium and hard problems is that the hard treasure hunt problem consists of multiple treasure goals to be achieved and the arrow boxes are very adjacent and dense. It is highly possible to choose a neighbour arrow in the opposite direction and traverse a totally wrong and disorganised path.

One way to solve this problem is to get the neighbour arrow alignments with the current object. Linear regression is used to find the alignments here. The central idea is that the curve passing through the centroid and yellow pixel of the current arrow can denote the alignment/direction of other pixels to the current bounding box. The bounding box with minimal regression error along with a smaller euclidean distance would be the target neighbour object. The main steps are

1. The main idea and flow are shown in Algorithm 5. In this case, there is an extra blue box which is not needed. Hence, this blue box is ignored and removed from the initial labelled components.

2. The arrow finding and red arrow identification is the same as Section 4.1. However, in this scenario, the treasure object ids are also calculated.

3. The arrow objects are traversed and the next arrow objects are chosen with both Euclidean distance and minimum regression error criteria. However, the treasure objects are traversed with only euclidean distance because they do not have yellow pixels. The algorithm is shown in Algorithm 6.

4. The search continues until two treasure items have been reached.

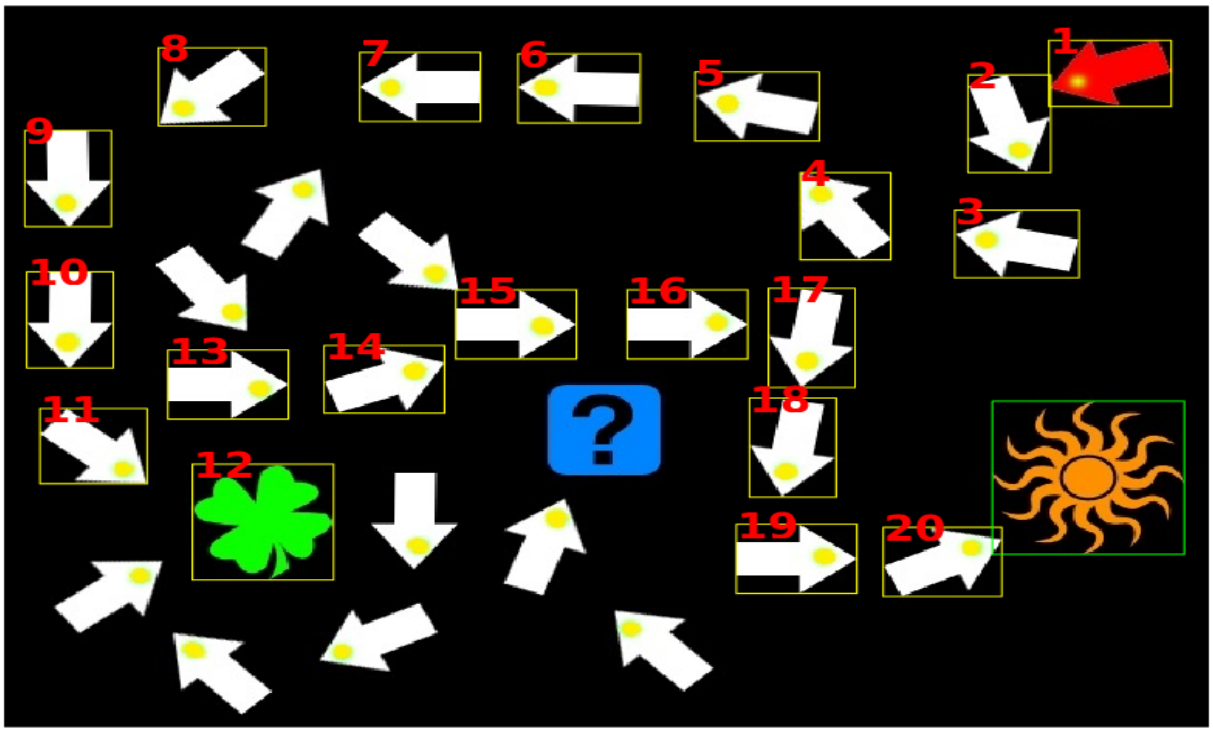


Figure 18: Hard Treasure Hunt path discovery (treasure green leaf and yellow sun)

The Figure 18 shows the path discovery in the hard treasure hunt problem. The main algorithm is in Algorithm 5. The object finder function is described in Algorithm 6 and relevant code snippet is also give as the MATLAB function *nextObjectFinder2*.

Algorithm 5 Treasure hunt Algorithm main structure.(Approach 2)

```
1: procedure treasureHard(image)
2:     Initialise system parameters                ▷ Read all the input values and initialise the params.
3:     im ← read(image)                            ▷ Read the image.
4:     binImage ← binarise(im)                     ▷ Binarisation with threshold (0.08).
5:     component ← componentLabel(binImage)        ▷ Extract connected components.
6:     properties ← regionProperty(component)      ▷ Extracting different comp properties.
7:     Draw bounding boxes for visualisation.
8:     blueBoxId ← FindBlueBox(properties, components)   ▷ Gets the BlueBox ID.
9:     component ← component − blueBoxId           ▷ Remove the bluebox with ? because that is not a treasure.
10:    arrowIds ← FindArrow(properties, components)      ▷ Gets the component ID of arrows.
11:    treasureIds ← component − arrowIds          ▷ Get the treasure object Ids
12:    redArrowId ← RedArrow(im, arrowIds, properties)   ▷ Gets the red arrow component.
13:    currentId ← redArrowId                      ▷ Initial state or starting state.
14:    path ← startId
15:    while reach both the treasure objectIds do
16:        currentId ← nextObjectFinder2(im, currentId, properties, treasureIds, path)   ▷ Find neighbour object
17:        path ← currentId                        ▷ Assign the new destination in the path.
18:    End Loop
19:    Visualise the path with path and im         ▷ Final Visualisation
```

Algorithm 6 Treasure hunt nextObjectFinder2 function (approach 2)

1: **procedure** NEXTOBJECTFINDER2(*im*, *currentId*, *properties*, *treasureIds*, *path*)
2: Initialise system parameters ▷ Read all the input values and initialise the params.
3: **if** *currentId* is a member of *treasureIds* **then** ▷ If current object is treasure object.
4: *currentId* ← *treasureNeighbour*(*im*, *currentId*, *properties*, *treasureIds*, *path*) ▷ Find next neighbour after a treasure object (similar with Algorithm 4).
5: **return** *currentId*
6: *currentCentroid* ← *properties*(*currentId*).*Centrooid* ▷ Get the centroid of the current arrow object
7: *pixels* ← *properties*(*currentId*).*PixelList* ▷ Get the list of pixels in current arrow object
8: *currentYellowPixel* ← **yellowPixelFinder**(*im*, *pixels*) ▷ Get a yellow pixel coordinate in the current arrow.
9: Fit a straightline (linear regression) between *currentCentroid*, *currentYellowPixel*.
10: *coefficients* ← *FIT*(*currentCentroid*, *currentYellowPixel*)
11: Initialise a *ObjectDistancePair* matrix ▷ Initialise a matrix to keep the object distance pair values
12: *minDistance* ← 200 ▷ Set a big value to the initial minimum neighbour distance.
13: *threshold* ← 30
14: **for** objId=1,2,..number of total objects **do** ▷ Run loop for all the objects.
15: **if** objId not in path **then** ▷ If the current object is not already in the path.
16: *tempCentre* ← *properties*(*currentId*).*Centrooid* ▷ Get the centroid of the neighbour
17: Calculate centroid distance between the current object and neighbour
18: *distance* ← *DISTANCE*(*currentCentroid*, *tempCentre*) ▷ euclidean/manhattan distance
19: **if** *minDistance* > *distance* **then**
20: *minDistance* ← *distance* + *threshold*
21: *ObjectDistancePair* ← *objId*, *distance* ▷ Assign the current nearest objId as the next neighbour
22: **End Loop**
23: Sort *ObjectDistancePair* based on *distance* and take 3 smallest distances with their Ids
24: **for** id = 1,2,3 **do**
25: **if** *treasureIds* are among the 3 nearest objects **then**
26: Then return *treasureIds* as *currentId*
27: *tmpcentroid* ← *properties*(*id*).*Centroid* ▷ Get the list of pixels in current arrow object
28: *tmppixels* ← *properties*(*id*).*PixelList* ▷ Get the list of pixels in current arrow object
29: *tmpYellowPixel* ← **yellowPixelFinder**(*im*, *tmppixels*) ▷ Get a yellow pixel coordinate in the arrow.
30: **Calculate the straight line regression loss of the neighbour centroid and yellow pixel**
31: *loss* ← *LINEFIT*(*coefficients*, [*tmpcentroid*, *tmpYellowPixel*])
32: The neighbour with least loss is more likely inclined with the current arrow object.
33: *currentId* ← *leastLoss*(*loss*, *id*)
34: **End Loop**
35: return *currentId*

```
  function cur_object = nextObjectFinder2(im, cur_object, props, n_objects,arrow_ind, treasures, path)
% if the current object is a treasure object
if ismember(cur_object,treasures)
    cur_object = nextObjectAfterTreasure(im, cur_object, props, n_objects,arrow_ind, treasures, path);
    return
end

% if the current object is an arrow then
% get centroid
current_centroid = props(cur_object).Centroid;
current_yellowPixel = current_centroid;
% get total pixel list in that object
current_pixelList = props(cur_object).PixelList;
% get yellow pixel location
current_yellowPixel = yellowPixelFinder(im,current_pixelList);
% fit a line through the centroid and yellow line
coeff = polyfit([current_centroid(1), current_yellowPixel(1)],[current_centroid(2), current_yellowPixel(2)],1);

% initalise a objectid-distance matrix for keeping the values
object_distance = zeros(30,2);
for i=1:30
  for j=1:2
      object_distance(i,j) = 99999 ;
  end
end
counter = 1;
min_centroid_distance = 200;
\% centroid calculation correction threshold
threshold = 30;
for object_id = 1 : n_objects
    if ismember(object_id,path) ~= true
      temp_centroid = props(object_id).Centroid;
      centroid_distance = (abs(current_centroid(1) - temp_centroid(1))+abs(current_centroid(2) - temp_centroid(2)));
      if min_centroid_distance > centroid_distance
          min_centroid_distance = centroid_distance + threshold;
          object_distance(counter,2) = round(centroid_distance);
          object_distance(counter,1) = object_id;
          counter = counter+1;
      end
    end
end
% sort the object-distance matrix based on the distance
object_distance = sortrows(object_distance,2);
% take nearest two arrow objects which have not been traversed before
object_distance = object_distance(1:3,:);
%disp(object_distance(1:5,:));
residual = 99999;

% get the line fitting between the current centroid and yellow pixel
% The fit the nearest arrow object centroids to get the right alignment
for loop = 1:3
    object_id = object_distance(loop,1);
    corr=distanceThresholdCheck(object_distance);
    % if a treasure object is nearby then no need to traverse further
    if ismember(object_id, treasures)
        cur_object = object_id;
        break;
    end
    if corr
```

```
          cur_object = object_id;
          return;
      end
      % linear regression error for finding the probable alignment
      if object_id ~= 99999
         centroid = props(object_id).Centroid;
         pixellist = props(object_id).PixelList;
         tmp_yellowPixel = yellowPixelFinder(im,current_pixelList);
         yFit = polyval(coeff,centroid(1));
         res_sum = sum(abs(yFit-centroid(2)));
         yFit = polyval(coeff,tmp_yellowPixel(1));
         res_sum = res_sum + sum(abs(tmp_yellowPixel(2)));
         disp(sprintf("%f obj -> error %f",object_id,res_sum));
         if residual > res_sum
             residual = res_sum;
             cur_object = object_id;
         end
      end
  end
end

% function: nextObjectAfterTreasure
    function cur_object = nextObjectAfterTreasure(im, cur_object, props, n_objects,arrow_ind, treasures, path)
    current_centroid = props(cur_object).Extrema(1,:);
    counter = 1;
    min_centroid_distance = 200;
    threshold = 30;
    for object_id = 1 : n_objects
        if ismember(object_id,path) ~= true
            temp_centroid = props(object_id).Centroid;
            centroid_distance = norm(abs(current_centroid(1) - temp_centroid(1))+abs(current_centroid(2) - temp_centroid(2))
                if min_centroid_distance > centroid_distance
                    min_centroid_distance = centroid_distance ;
                    cur_object = object_id;
                end
        end
    end
end

 % yellow pixel finder
function block_yellow_pixel = yellowPixelFinder(im,current_pixelList)
    pixels = length(current_pixelList);
    i = 1;
    yellow_pixel = [];
    while i < pixels
        % extract colour of the centroid point of the current arrow
        pixel_colour = im(round(current_pixelList(i,2)), round(current_pixelList(i,1)), :);
        if pixel_colour(:, :, 1) > 240 && pixel_colour(:, :, 2) > 230 && pixel_colour(:, :, 3) > 15
            yellow_pixel = [yellow_pixel;current_pixelList(i,:)];
        end
        i = i+1;
    end
    %   disp(yellow_pixel)
    block_yellow_pixel = mean(yellow_pixel,1);
    %   disp(current_yellowPixel);
end
```

4.3 Discussion

Some of the challenges which occurred during the implementation are

1. The initial threshold value is very important as it changes the centroid pixel position while measuring the properties of the bounding box. In this experiment, it is set to 0.08.

2. The congested arrows make the path traversal prone to errors such as choosing an arrow block in the wrong direction. Hence, there is a chance of taking a totally wrong path in the next steps.

3. Euclidean distance has its problems while measuring the distance between neighbour centroids because it does not consider the direction/alignment of the current object.

4. There are multiple yellow pixels in the yellow regions. Different yellow pixels and centroid fit different regression curves and, as a result, different alignment representations.

5. In these tasks, the arrows are chosen using the area attributes because it uses an already computed attribute. A yellow pixel finder function is also implemented to find if the box contains yellow pixes and thus find arrows. However, yellow pixel-based arrow detection takes a significantly large number of computation loops. So the area threshold 1600 is used to separate arrow and non-arrow objects.

6. Practically, the centroid calculation has minor noise, which can be removed by using a threshold distance (Algorithm 6).

5 Task 5: Siamese CNN, YOLO CNN, Capsule CNN

Siamese CNN, YOLO CNN and Capsule CNN are based on convolutional neural layers. In this section, firstly, I discuss the CNN layers briefly (Section 5.1) along with a general CNN model, in Section 5.2,5.3, 5.4 the intuitions and working mechanisms of siamese, YOLO and capsule networks have been discussed. The discussion is mainly about how these models have evolved from CNNs and how they differ in the working principles and architecture diagrams from the literature. Finally, in Section 5.5, these models have been compared with some aspects while stretching the discussions furthermore from the previous sections.

5.1 CNN

The Convolutional neural network (CNN) topology is gridlike with overlapping kernel. The goal for overlapping kernal is parameter sharing and shared layers [2]. The biological

inspiration for weight sharing and convolutions among neurons were seen in [3] and [4]. In computational modelling, the term 'convolution' comes from the mathematical operation convolution, which operates on two functions to produce a resulting function producing the overlap of one function shift over another. If $m(x)$ and $n(x)$ are two functions over a continuous variable y, the convolution over an infinite interval would be

$$f(\mathbf{x}) * g(\mathbf{x}) = \int_{-\infty}^{+\infty} f(\gamma) \times g(\mathbf{x} - \gamma) d\gamma, \tag{1}$$

where γ is the continuous time step, $*$ is convolution operator. In signal processing, $n[y]$ is the impulse function over $m[y]$. In image processing, the interval is finite. The $n[y]$ is the local receptive field that shares same set of weights, on different regions of the image $m[y]$, in which the layers extract visual features and combine the set of outputs to form feature maps. Kernels of size $[p \times q \times T]$ ([height × width × depth], and $t = 1, 2, \cdots, T$) are used, the t^{th} convolutional feature map can be denoted as:

$$y_t = R\left(\sum_k n_k * y_k\right), \tag{2}$$

where n_k is the k^{th} kernel and y_k ($k = 1, 2, \cdots, K$) is the k^{th} input feature map of size $[A \times B]$ and $R(\cdot)$ is the nonlinear activation function to convert the output of convolution into nonlinear for complex problem space ([5, 6]).

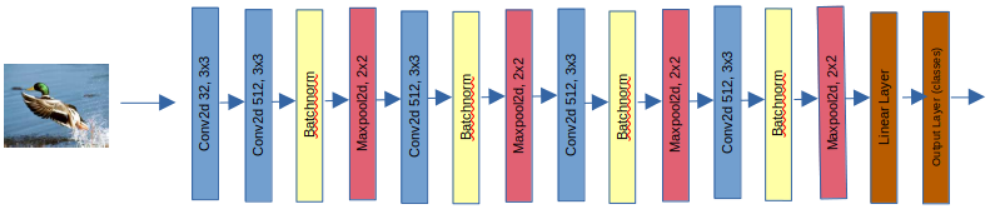


Figure 19: A general CNN architecture comprising several layers

Generally, CNNs have several the convolution layers and spatial/temporal sub-sampling layers along with normalisation layers ([2])(an example shown in Figure 19). The shared regions, sparsed interactions, and equivariant representations ([7]) allow the CNNs to work with variable-length data for better representation. As a result, the convolution process parameter sharing has better efficiency than the standard fully-connected neural layers.

5.2 Siamese CNN

The Siamese CNN (S-CNN) architectures use similar CNN layers and models with an additional feature, which is training the network to learn embedding difference among similar objects or object groups [8]. The S-CNN is trained using pairwise data with the same model and the pairs are similar or dissimilar. Generally, the difference or margin is learnt by the model using either contrastive loss or triplet loss. A general model architecture is shown in Figure 20. The final layer of the model learns the difference between the embedding with the two images. The important thing to remember here is that in both samples same network model has been used to extract the embedding.

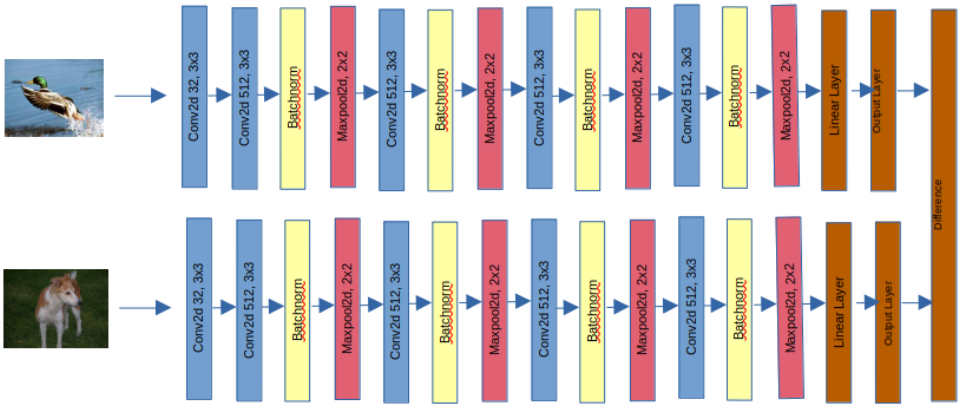


Figure 20: A general CNN siamese architecture comprising triplet/contrastive loss

The backbone of the S-CNN architectures are loss functions. Two of the most used loss functions are discussed here and their relevance to the siamese networks.

5.2.1 Contrastive Loss

The goal of contrastive loss is to signify how similar or dissimilar the data pairs are. Hence, training the model with finer representation learning in the layers. The model behaviour is discriminatively trained for separate behaviour with similar or dissimilar data. The general form of contrastive loss is given below.

$$L(\theta, (S, \vec{X_1}, \vec{X_2})) = (1 - S)L_{similar}(Distance_\theta) + SL_{dissimilar}(Distance_\theta) \quad (3)$$

where θ is the model, and L is the loss function. S signifies if the sample pair is similar or dissimilar. $L_{dissimilar}, L_{similar}$ can be two loss functions in two scenarios. The $Distance_\theta$ is a function measuring some form of embedding distance.

5.2.2 Triplet Loss

The problem with contrastive loss is that, while training with contrastive loss, the model has no knowledge about the data pairs. This problem can risk failing to learn the appropriate discriminative local feature representations. One approach to come across this problem is to have a sample to which the pair is compared. This sample is called the anchor sample or anchor data. One positive and one negative sample is given along with the anchor data. The goal is to achieve a non-greedy way of learning generalised representations about similar or dissimilar pairs. The general equation of triplet loss is given as

$$L(\theta, (S, \vec{X_a}, \vec{X_p}, \vec{X_n})) = \sum_{n}^{N} [\, ||f(X_a) - f(X_p)||_2^2 - ||f(X_a) - f(X_n)||_2^2] \tag{4}$$

Where X_a is the anchor sample, X_p is the positive and X_n is the negative sample. The total number of samples are N. The loss function tends to maximise the distance between anchor and the negative sample and minimise the distance between the positive sample and the anchor sample.

5.3 YOLO CNN

Contrary to the conventions CNN model approaches where sliding window convolutions are used to build context representation for the image for classification, YOLO or *You Only Look Once* approach uses multiple bounding boxes and classify segments and train the image on all the combined loss and bounding box coordinates. Rather than a straightforward classification approach, it treats the task as a regression problem [9, 10]. In this way it learns the objects and where the objects are resided in the image.

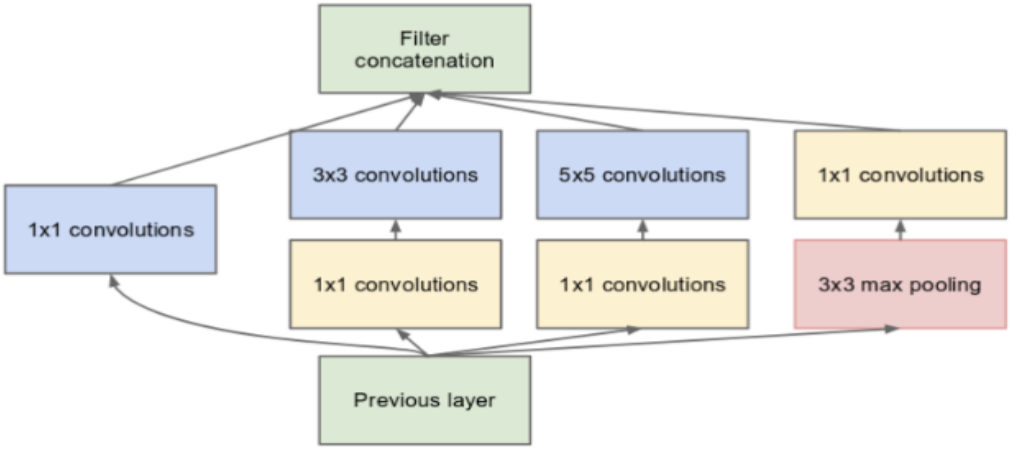


Figure 21: Typical GoogleNet block (blog.paperspace.com) base for YOLO models

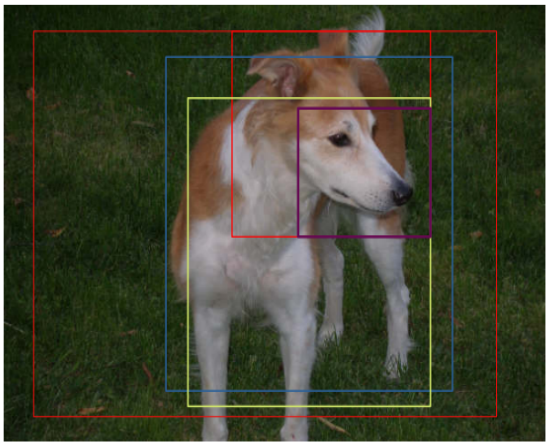


Figure 22: A representation of YOLO perspective of several glances/instances of a image

It learns the bounding boxes by training with anchor boxes with the convolution layers [10]. YOLOv1 trains only with the bounding boxes, but YOLOv2 uses the anchor boxes. The advantage is that v2 learns better representation and localised representation, but it also loses some accuracy. The location of the boxes or the prior is chosen using k-means dimensional clustering where big boxes are chosen to minimise error with bigger boxes than to choose smaller boxes. The main architecture of YOLOv2 comes from the GoogleNet (Figure 21) architecture which ensures faster training rather than the VGG16 model [10].

5.4 Capsule CNN

A capsule in Capsule CNN is generally a set of convolution layers concatenated together. Each CNN layer learns different abstract spatiotemporal representations, and the output matrices of these convolution units are concatenated to form a capsule, which is normalised between $\mathbf{0}$ and $\mathbf{1}$ using a squashing function. The output of a capsule n, $\mathbf{o}_n^t$, is

$$\mathbf{o}_n^t = \mathbf{squash}(\mathbf{y}_n), \tag{5}$$

where $\mathbf{y}_n$ is the input to the capsule n and $\mathbf{squash}(.)$ is a squashing function. In [11], $\mathbf{squash}(\cdot)$ is defined as follows

$$\mathbf{squash}(\mathbf{x}) = \frac{\|\mathbf{x}\|^2}{\mathbf{1} + \|\mathbf{x}\|^2} \frac{\mathbf{x}}{\|\mathbf{x}\|}. \tag{6}$$

These groups of capsule-CNN's jointly learn localised representations and output a vector. Each capsule learns a specific spatial/temporal characteristic representation of the data. The capsule CNN preserves this representation by not applying pooling layers and it learns hierarchical relationships between the layers based on this layer-wise representation. It learns these hierarchical relations using routing mechanism [11]. *Routing by agreement*

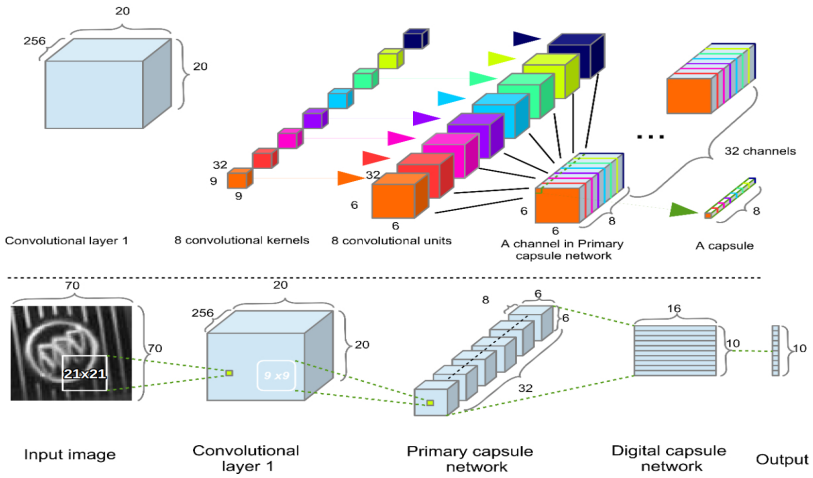


Figure 23: A representation of CapsuleCNN as shown in [1].

algorithm uses the previous layer capsules to predict the output in the next capsule layers. The input of the higher layer capsule $\mathbf{n}$, $\mathbf{y_n}$ is

$$\mathbf{y_n} = \sum_{\mathbf{m}} \mathbf{W_{mn}} \hat{\mathbf{Pr}}_{\mathbf{n|m}}, \tag{7}$$

here $\hat{Pr}_{n|m}$ is the prediction of capsule m for the capsule n and coefficients $W_{m|n}$ are weights. The weights are computed through *routing-by-agreement*. The agreement depend on how close the predictions are and those hierarchical capsule have bigger weights connecting them. Therefore, the weights between m^{th} capsule on the n^{th} capsule is dynamically adjusted based on how related the lower layer capsule is with the upper layer capsule. Hence forming a hierarchical context learning mechanism.

5.5 Comparison and Application

5.5.1 Model size

The capsule CNN is the most complex and computationally complex among these three types of model architectures. The reason is the siamese CNN and Yolo CNN both use CNN architectures such as GoogleNet, VGGNet, Inception etc [10]. However, capsule CNN squeezes multiple CNNs in one capsule in a layer and in an ideal scenario, it uses many capsules and layers [11]. Hence, for training similar sized corpora with comparable results, capsule networks will use significantly more parameters than S-CNN and YOLO CNN. Also, capsule networks are expensive for training models due to the huge number of parameters and internal routings.

5.5.2 Representation Learning

The nature of representation learning is different in these three models. In siamese network, the model has no prior knowledge about the pairs, and it performs greedy learning. Thus the middle layers are prone to learn less local representation [8]. Different gating mechanisms between layers have been proposed, and a triplet loss function to resolve this. Yolov1 [9] model also suffers from learning less local information. But it is improved on Yolov2 [10]. However, capsules perform hierarchical learning with *routing by agreement*, which allows the context to build up from lower label capsule representation (local context) to higher level capsule representation (global context). This global context modelling with local context is explicit in capsule CNN. Siamese networks and capsule networks learn representation differences in similar data and thus they are less prone to adversarial attacks.

5.5.3 K-shot Learning

Siamese CNNs enable K-shot learning where in the pair-wise training the model learns the representational difference of the new object faster with contrastive loss and triplet loss (Section 5.2). YOLO network learns object representation in accordance with the global context of the object in the image. However, it has problems with learning smaller objects such as a flock of birds etc [9]. Capsule CNNs are the least flexible in terms of adapting to new data and keeping the old learned representation stable.

5.5.4 Application

Due to the computational complexity and model size issues, capsule networks are not generally used to train large corpora or tasks requiring huge data. However, in smaller corpora or tasks, such as handwriting recognition, sign language recognition, medical image classification, capsule networks have shown the state of the art results [11, 1]. For bigger corpora such as COCO, WordNet, CIFAR, capsules use significantly higher training time compare to S-CNN and YOLO.

S-CNNs learn discriminative features among objects with comparably faster execution time. So, they are usually used in object tracking, segmentation and detection tasks. Also, S-CNNs are used in image security applications such as signature verification, retina verification etc. The fastest model among these three is the YOLO model. It can execute 45 frames per second and is incredibly fast for identifying objects in images. YOLO is vastly used in image segmentation, captioning, object detection and autonomous driving.

References

[1] R. Chen, M. A. Jalal, L. Mihaylova, and R. K. Moore, "Learning capsules for vehicle logo recognition," in *2018 21st International Conference on Information Fusion (FUSION)*. IEEE, 2018, pp. 565–572.

[2] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2323, 1998.

[3] D. H. Hubel and T. N. Wiesel, "Receptive fields of single neurones in the cat's striate cortex," *The Journal of Physiology*, vol. 148, no. 3, pp. 574–591, 1959.

[4] ——, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *The Journal of Physiology*, vol. 160, no. 1, pp. 106–154, 1962.

[5] V. Nair and G. E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," *Proceedings of the 27th International Conference on Machine Learning*, no. 3, pp. 807–814, 2010.

[6] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical Evaluation of Rectified Activations in Convolution Network," *ICML Deep Learning Workshop*, pp. 1–5, 2015.

[7] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[8] R. R. Varior, M. Haloi, and G. Wang, "Gated siamese convolutional neural network architecture for human re-identification," in *European conference on computer vision*. Springer, 2016, pp. 791–808.

[9] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[10] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7263–7271.

[11] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic routing between capsules," in *Advances in Neural Information Processing Systems*, 2017, pp. 3859–3869.

6 Appendix

6.1 Code for problem 2

```
% Task 2.1 code
```

```
    % GingerbreadMan.m
    % clear all

clear all;

% define bbox



fist_image = imread("GingerBreadMan_first.jpg");
second_image = imread("GingerBreadMan_second.jpg");

fist_image_gray = rgb2gray(fist_image);
second_image_gray = rgb2gray(second_image);

corner_points = 100;
cornerPoints = show_corners(fist_image_gray,corner_points,true);

h = figure;
movegui(h);
hViewPanel = uipanel(h,'Position',[0 0 1 1],'Title','Plot of Optical Flow Vectors');
hPlot = axes(hViewPanel);

opticFlow = opticalFlowLK('NoiseThreshold',0.009);
flow = estimateFlow(opticFlow,fist_image_gray);

imshow(fist_image);
hold on
plot(flow,"DecimationFactor",[5 5],"ScaleFactor",10,"Parent",hPlot);
hold off
pause(10^-1);


flow = estimateFlow(opticFlow,second_image_gray);
imshow(second_image);
hold on
plot(flow,"DecimationFactor",[5 5],"ScaleFactor",10,"Parent",hPlot);
hold off
pause(10^-3);

%%%%%%%%%%%%%%%%%%%%%% function show_corners.m %%%%%%%%%%%%%%%%

function cornerPoints=show_corners(gray_image,num_corners,visualise)

cornerPoints  = corner(gray_image,num_corners);
if visualise == true
```

```
    imshow(gray_image);
    hold on
    plot(cornerPoints(:,1),cornerPoints(:,2),"*","Color","c")
    hold off
    % pause(5);
    % imshow(cornerPoints);
end

%%%%%%%%%%%%%%%%%% Task 2.2 %%%%%%%%%%%%%%%%%%%%%%%%

% clear all

clear all;

% define rectangle

input_bbox = [245 225 35 35];

% load ground truth
ground_truth = load("red_square_gt.mat").gt_track_spatial;

% read first reference frame
video = VideoReader('red_square_video.mp4');
fist_frame = read(video,1);
fist_frame_gray = rgb2gray(fist_frame);

% corner params
num_corners = 4;
cornerPoints = show_corners(fist_frame_gray,num_corners,false);
pointofInterest = cornerPoints(1,:);
corner_x = pointofInterest(1);
corner_y = pointofInterest(2);

% video reader object
videoReader = VideoReader('red_square_video.mp4');

% optical flow object
opticFlow = opticalFlowLK('NoiseThreshold',0.009);
flow = estimateFlow(opticFlow,fist_frame_gray);
h = figure;
movegui(h);
hViewPanel = uipanel(h,'Position',[0 0 1 1],'Title','Plot of Optical Flow Vectors');
hPlot = axes(hViewPanel);
counter = 0;
SE = 0;
errors = [];
while hasFrame(videoReader)
```

```
    distance = 99999; %assign a max distance
    frameRGB = readFrame(videoReader);
    frameGray = im2gray(frameRGB);
    cornerPoints = corner(frameGray,num_corners);
    counter = counter+1;
    % calculate shortest neighbour of the previous corner point
    for i=1:length(cornerPoints)
        tmp_distance = norm(cornerPoints(i,:)-[corner_x corner_y]);
        % take smaller distance point
        if distance > tmp_distance
            distance = tmp_distance;
            corner_x_temp = cornerPoints(i,1);
            corner_y_temp = cornerPoints(i,2);

        end
    end
    corner_x = corner_x_temp;
    corner_y = corner_y_temp;
    % optical flow at that given frame
    flow = estimateFlow(opticFlow,frameGray);
    % adjust new points
    x_new = corner_x + flow.Vx(round(corner_y), round(corner_x));
    y_new = corner_y + flow.Vy(round(corner_y), round(corner_x));
    corner_x = x_new;
    corner_y = y_new;
    % shift the bounding box
    input_bbox = ShiftBbox(input_bbox,[corner_x corner_y]);

    % show the current frame, bounding box and optical flow direction
    imshow(frameRGB);
    hold on
    rectangle("Position",input_bbox);
    hold on
    plot(flow,'DecimationFactor',[5 5],'ScaleFactor',10,'Parent',hPlot);
    hold on
%     distance = norm(ground_truth(counter,:)-[corner_x corner_y]);
    distance = sqrt(mean(ground_truth(counter,:)-[corner_x corner_y]).^2);
    errors = [errors;distance];
    gt_x = ground_truth(counter,1);
    gt_y = ground_truth(counter,2);
    text_gt = ['Ground truth:' num2str(gt_x) ',' num2str(gt_y)];
    text_pr = ['Predicted:' num2str(corner_x) ',' num2str(corner_y)];
    text_error = ['SE:' num2str(distance)];
    txt = {text_gt,text_pr,text_error};
    text(100,50,txt)
    hold off
    SE = SE + sqrt(mean(ground_truth(counter,:)-[corner_x corner_y]).^2);
```

```
    pause(10^-3)
end

% divide the total distance by number of frames to get the avarage square error
SE = SE./counter

figure
plot(errors,'Color',[0,0.7,0.9])

title('2-D Line Plot')
xlabel('Frames')
ylabel('Error')
```

6.2 Problem 4 code

6.2.1 Problem 4 Easy Medium Scenario

```
%%%%%%%%%%%%%%%%%% 4.1,4.2 Easy, medium %%%%%%%%%%%%%%%

%% main.m
close all;
clear all;
%% Reading image

im = imread('Treasure_medium.jpg'); % change name to process other images
imshow(im);
% pause;
%% Binarisation

bin_threshold = 0.09; % parameter to vary 0.08
bin_im = im2bw(im, bin_threshold);
imshow(bin_im);
% pause;
%% Extracting connected components

con_com = bwlabel(bin_im);
imshow(label2rgb(con_com));
% pause;
%% Computing objects properties

props = regionprops(con_com, rgb2gray(im), "Centroid","Area","BoundingBox","Orientation","PixelList"
%% Drawing bounding boxes

n_objects = numel(props);
imshow(im);
hold on;
```

```
for object_id = 1 : n_objects
    rectangle('Position', props(object_id).BoundingBox, 'EdgeColor', 'b');
end
\Stahold off;
% pause;
%% Arrow/non-arrow determination
% You should develop a function arrow_finder, which returns the IDs of the arror
% objects. IDs are from the connected component analysis order. You may use any
% parameters for your function.

arrow_ind = arrow_finder(props,n_objects);
disp(arrow_ind);
%% Finding red arrow

n_arrows = numel(arrow_ind);
start_arrow_id = 0;
% check each arrow until find the red one
for arrow_num = 1 : n_arrows
    object_id = arrow_ind(arrow_num);    % determine the arrow id

    % extract colour of the centroid point of the current arrow
    centroid_colour = im(round(props(object_id).Centroid(2)), round(props(object_id).Centroid(1)), :
    if centroid_colour(:, :, 1) > 240 && centroid_colour(:, :, 2) < 10 && centroid_colour(:, :, 3) <
        % the centroid point is red, memorise its id and break the loop
        start_arrow_id = object_id;
        break;
    end
end
%% Hunting

cur_object = start_arrow_id; % start from the red arrow
path = cur_object;

% while the current object is an arrow, continue to search
while ismember(cur_object, arrow_ind)
    disp(path);
    % You should develop a function next_object_finder, which returns
    % the ID of the nearest object, which is pointed at by the current
    % arrow. You may use any other parameters for your function.

    cur_object = nextObjectFinder(im, cur_object, props, n_objects, arrow_ind, path);
    path(end + 1) = cur_object;
end
%% visualisation of the path

imshow(im);
hold on;
```

```
for path_element = 1 : numel(path) -1
    object_id = path(path_element); % determine the object id
    rectangle('Position', props(object_id).BoundingBox, 'EdgeColor', 'y');
    str = num2str(path_element);
    text(props(object_id).BoundingBox(1), props(object_id).BoundingBox(2), str, 'Color', 'r', 'FontW
end

% visualisation of the treasure
treasure_id = path(end);
rectangle('Position', props(treasure_id).BoundingBox, 'EdgeColor', 'g');

%%%%%%%%%% arrow_finder.m %%%%%%%%%%%%
function arrow_ind = arrow_finder(props,n_objects)
arrow_ind = [];
    for object_id = 1 : n_objects
        area = props(object_id).Area;
        centrod = props(object_id).Centroid;
        pixellist = props(object_id).PixelList;
        pixelValues = props(object_id).PixelValues;
        if area < 1600
            arrow_ind = [arrow_ind ; object_id];
    end

end

%%%%%%%%%%nextObjectFinder.m %%%%%%%%

function cur_object = nextObjectFinder(im, cur_object, props, n_objects,arrow_ind, path)
  current_centroid = props(cur_object).Centroid;
  min_centroid_distance = 99999;
  for object_id = 1 : n_objects
      if ismember(object_id,path) ~= true
        temp_centroid = props(object_id).Centroid;
        centroid_distance = norm(round(current_centroid) - round(temp_centroid));
        if min_centroid_distance > centroid_distance
            min_centroid_distance = centroid_distance;
            cur_object = object_id;
        end
      end
   end
end
```

6.2.2 Task 4 Hard Scenario

```
    %% treasure_hard.m%%%%%
close all;
clear all;
```

```
%% Reading image

im = imread('Treasure_hard.jpg'); % change name to process other images
imshow(im);
% pause;
%% Binarisation

bin_threshold = 0.08; % parameter to vary 0.09
bin_im = im2bw(im, bin_threshold);
imshow(bin_im);
% pause;
%% Extracting connected components

con_com = bwlabel(bin_im);
imshow(label2rgb(con_com));
% pause;
%% Computing objects properties

props = regionprops(con_com, rgb2gray(im), "Centroid","Area","BoundingBox","Orientation","PixelList"
%% Drawing bounding boxes

n_objects = numel(props);
imshow(im);
hold on;
for object_id = 1 : n_objects
    rectangle('Position', props(object_id).BoundingBox, 'EdgeColor', 'g');
end
hold off;
% pause;
%%  Blue box identification

% identifyning blue question mark box

blue_box_id = 0;
for obj_id = 1 : n_objects
    % disp(obj_id);
    % extract colour of the centroid point of the current arrow
    centroid_colour = im(round(props(obj_id).Centroid(2)), round(props(obj_id).Centroid(1)), :);
    if centroid_colour(:, :, 1) < 2 && centroid_colour(:, :, 2) < 2 && centroid_colour(:, :, 3) < 15
        % the centroid point is red, memorise its id and break the loop
        blue_box_id = obj_id;
        break;
    end
end

props(blue_box_id)=[];
n_objects = numel(props);
```

```
%%
% You should develop a function arrow_finder, which returns the IDs of the arror
% objects. IDs are from the connected component analysis order. You may use any
% parameters for your function.
%% *Arrow/non-arrow determination*

arrow_ind = arrow_finder(props,n_objects);
% disp(arrow_ind);

%% Finding red arrow

n_arrows = numel(arrow_ind);
start_arrow_id = 0;
% check each arrow until find the red one
for arrow_num = 1 : n_arrows
    object_id = arrow_ind(arrow_num);    % determine the arrow id

    % extract colour of the centroid point of the current arrow
    centroid_colour = im(round(props(object_id).Centroid(2)), round(props(object_id).Centroid(1)), :
    if centroid_colour(:, :, 1) > 240 && centroid_colour(:, :, 2) < 10 && centroid_colour(:, :, 3) <
        % the centroid point is red, memorise its id and break the loop
        start_arrow_id = object_id;
        break;
    end
end
treasures = [];
for i = 1:n_objects
    if ismember(i,arrow_ind)~=true
        treasures = [treasures;i];
    end
end
%% Hunting

cur_object = start_arrow_id; % start from the red arrow
path = cur_object;
% while the current object is an arrow, continue to search
found = 0;
while ismember(cur_object, arrow_ind) || found < 2
    disp(path);
    % You should develop a function next_object_finder, which returns
    % the ID of the nearest object, which is pointed at by the current
    % arrow. You may use any other parameters for your function.

    cur_object = nextObjectFinder(im, cur_object, props, n_objects, arrow_ind, treasures, path);
    path(end + 1) = cur_object;
    if ismember(cur_object,treasures)
        found = found+1;
```

```
    end
end
%% visualisation of the path

imshow(im);
hold on;
for path_element = 1 : numel(path) -1
    object_id = path(path_element); % determine the object id
    rectangle('Position', props(object_id).BoundingBox, 'EdgeColor', 'y');
    str = num2str(path_element);
    text(props(object_id).BoundingBox(1), props(object_id).BoundingBox(2), str, 'Color', 'r', 'FontW
end

% visualisation of the treasure
treasure_id = path(end);
rectangle('Position', props(treasure_id).BoundingBox, 'EdgeColor', 'g');

%%%%%%% nextObjectFinder.m%%%%%%
function cur_object = nextObjectFinder(im, cur_object, props, n_objects,arrow_ind, treasures, path)
  % if the current object is a treasure object
  if ismember(cur_object,treasures)
      cur_object = nextObjectAfterTreasure(im, cur_object, props, n_objects,arrow_ind, treasures, pa
      return
  end

  % if the current object is an arrow then
  % get centroid
  current_centroid = props(cur_object).Centroid;
  current_yellowPixel = current_centroid;
  % get total pixel list in that object
  current_pixelList = props(cur_object).PixelList;
  % get yellow pixel location
  current_yellowPixel = yellowPixelFinder(im,current_pixelList);
  % fit a line through the centroid and yellow line
  coeff = polyfit([current_centroid(1), current_yellowPixel(1)],[current_centroid(2), current_yellowP

  % initalise a objectid-distance matrix for keeping the values
  object_distance = zeros(30,2);
  for i=1:30
    for j=1:2
        object_distance(i,j) = 99999 ;
    end
  end
  counter = 1;
  min_centroid_distance = 200;
  % centroid calculation correction threshold
  threshold = 30;
```

```
  for object_id = 1 : n_objects
      if ismember(object_id,path) ~= true
        temp_centroid = props(object_id).Centroid;
        centroid_distance = (abs(current_centroid(1) - temp_centroid(1))+abs(current_centroid(2) - te
        if min_centroid_distance > centroid_distance
            min_centroid_distance = centroid_distance + threshold;
            object_distance(counter,2) = round(centroid_distance);
            object_distance(counter,1) = object_id;
            counter = counter+1;
        end
      end
  end
  % sort the object-distance matrix based on the distance
  object_distance = sortrows(object_distance,2);
  % take nearest two arrow objects which have not been traversed before
  object_distance = object_distance(1:3,:);
  %disp(object_distance(1:5,:));
  residual = 99999;

  % get the line fitting between the current centroid and yellow pixel
  % The fit the nearest arrow object centroids to get the right alignment
  for loop = 1:3
      object_id = object_distance(loop,1);
      corr=distanceThresholdCheck(object_distance);
      % if a treasure object is nearby then no need to traverse further
      if ismember(object_id, treasures)
          cur_object = object_id;
          break;
      end
      if corr
          cur_object = object_id;
          return;
      end
      % linear regression error for finding the probable alignment
      if object_id ~= 99999
         centroid = props(object_id).Centroid;
         pixellist = props(object_id).PixelList;
         tmp_yellowPixel = yellowPixelFinder(im,current_pixelList);
         yFit = polyval(coeff,centroid(1));
         res_sum = sum(abs(yFit-centroid(2)));
         yFit = polyval(coeff,tmp_yellowPixel(1));
         res_sum = res_sum + sum(abs(tmp_yellowPixel(2)));
%          disp(sprintf("%f obj -> error %f",object_id,res_sum));
         if residual > res_sum
             residual = res_sum;
             cur_object = object_id;
         end
```

```
      end
  end
end

%%%%%%yellowPixelFinder.m%%%%%%
function block_yellow_pixel = yellowPixelFinder(im,current_pixelList)
    pixels = length(current_pixelList);
    i = 1;
    yellow_pixel = [];
    while i < pixels
        % extract colour of the centroid point of the current arrow
        pixel_colour = im(round(current_pixelList(i,2)), round(current_pixelList(i,1)), :);
        if pixel_colour(:, :, 1) > 240 && pixel_colour(:, :, 2) > 230 && pixel_colour(:, :, 3) > 15
            yellow_pixel = [yellow_pixel;current_pixelList(i,:)];
        end
        i = i+1;
    end
    %   disp(yellow_pixel)
    block_yellow_pixel = mean(yellow_pixel,1);
    %   disp(current_yellowPixel);
end
%%%%%nextObjectAfterTreasure.m%%%%%%%%%
function cur_object = nextObjectAfterTreasure(im, cur_object, props, n_objects,arrow_ind, treasures,
    current_centroid = props(cur_object).Extrema(1,:);
    counter = 1;
    min_centroid_distance = 200;
    threshold = 30;
    for object_id = 1 : n_objects
        if ismember(object_id,path) ~= true
            temp_centroid = props(object_id).Centroid;
            centroid_distance = norm(abs(current_centroid(1) - temp_centroid(1))+abs(current_centroi
                if min_centroid_distance > centroid_distance
                    min_centroid_distance = centroid_distance ;
                    cur_object = object_id;
                end
        end
    end
end
%%%%%%%%distanceThresholdCheck.m%%%%%%%

function corr = distanceThresholdCheck(matrix)
if (matrix(2,2)-matrix(1,2)) == 75
    corr=true;
else
    corr=false;
end
end
```

```
%%%%%%%%arrowFinder.m%%%%%%%%
function arrow_ind = arrow_finder(props,n_objects)
arrow_ind = [];
    for object_id = 1 : n_objects
        area = props(object_id).Area;
        centrod = props(object_id).Centroid;
        pixellist = props(object_id).PixelList;
        pixelValues = props(object_id).PixelValues;
        if area < 1600
            arrow_ind = [arrow_ind ; object_id];
    end

end
```