Hungarian Auctioning - Lab 7.4

Recap

This is the Lab on using a Hungarian Auctioning in CE6003's Object Tracking. You should complete the tasks in this lab as part of the Tracking Multiple Objects / Hungarian Auctioning section of the lesson.

Please remember this lab must be completed before taking the quiz at the end of this lesson.

First, if we haven't already done so, we need to clone the various images and resources needed to run these labs into our workspace.

In [1]:

```
#!git clone https://github.com/mcnamarad1971/CE6003.git
#!git clone https://github.com/EmdaloTechnologies/CE6003.git
```

Program Description

This program demonstrates a very simple 'allocation' mechanism based on a Hungarian auction. We're using a tracker derived from a Kalman filter, the filter we developed in the Kalman lab. We're going to use Yolo3 for detections - as usual, and we're going to demonstrate tracking multiple objects (people) using this trio of Hungarian, Kalman, Yolo3.

Purpose

The purpose of this lab is to demonstrate how we might use a Hungarian Algorithm for tracking multiple objects in a video stream, in conjunction with a detector and a set of trackers.

Overall Algorithm

Based on detections in a previous image:

- run a simple Kalman filter to make a prediction about the position of those objects in a new frame
- gather the detections for the new frame
- use a Hungarian algorithm and a cost function to allocate the new detections to two pools (reasonably optimally at reasonable computational cost)

    1. New detections that best match the old detection's Kalman loops
    2. Completely new detections that need a new Tracker/Kalman to manage them

- Finally the Trackers need to be pruned if they haven't had a new detection sufficiently recently

For demo purposes, we pretty print the trackers back onto the frames.

The Story So Far

To illustrate how to track something in a video stream, we have used the following technique to generate a set of images for you to work on.

What we did was we generated a short video - just recording a few people walking around, on an iPhone.

Then we used `ffmpeg` to decompose about 5 seconds of that video down into still images.

```
ffmpeg -i $vid_in -vf fps=30 people%03d.png
```

We saved those frames as `people%03d.png` in the git repository in the multiple-detections directory at `images/lab7/` . You should be able to access them at `/content/CE6003/images/lab7/multiple-objects`

We've run yolo3 over those frames to generate bounding boxes and saved those bounding boxes into the same directory.

The file format is comma-separated values and the values are as shown here:

frame index	object-type	centre-x	centre-y	width	height	confidence
int	-1	float	float	float	float	float

- The object type is always a person - that's all we inferred for.
- The centre-x and width are fractions of the image's width
- The centre-y and height are fractions of the image's height
- The confidence is supplied by Yolo3

What Happens Now

For each image in the directory, in order,

- we'll find the centre of the detection in that image (if any)
- we'll build a bounding box for the detection in that image
- we'' derive a variance term (crudely) from the Yolo confidence for that image

This time we'll supply three lists - a list of the centres of the bounding boxes in the frame, a list of the bounding boxes in the frame, and a list of the Yolo confidence terms.

These we'll supply to a Kalman tracker and a Hungarian allocation block.

Then, we'll explore how a the Hungarian allocator distributes the detections to Kalman-based trackers to track multiple objects in the image stream.

Imports

We're using standard imports along with some imports to help display our work on colab. The only thing to note is we're importing scipy's linear_sum_assignment operator. This will run the Hungarian auction for us.

In [2]:

```
import os
import math
import re
import io
import cv2
import numpy as np
from scipy.optimize import linear_sum_assignment
import time
import base64
from IPython.display import clear_output, Image, display
```

Major Tunable Parameters

This is where we tune the behaviour of our program.

There are a few things to look at here:

- `maxAge` - this is where we're setting the trade-off between treating a detection as part of an old series or starting a new series. All we do is simply say - if a tracker hasn't had a match for 4 frames, then get rid of it. If a detection comes in after that, then its a new Tracker.
- `minHits` - pretty typical - get tracked on the first hit.
- `ndThreshold` - a tuning parameter for the Hungarian. The Hungarian is typically greedy and matches as much as it can. We're just saying here that empirically there are matches of a quality that are too poor and we're not interested in them.
- `minBlockArea` - for demo purposes, it was a little clearer if we only tracked larger objects in the scene (approx 200 x 200 pixels)

In [3]:

```
#IMGDIR="/content/CE6003/images/lab7/multiple-objects/"
IMGDIR="./images/lab7/multiple-objects/"

# Find pngs and bounding boxes for pngs
pngDir = IMGDIR
bbDir = IMGDIR

# Initialise an OpenCV video writer object
writer = None


maxAge = 4 # number of consecutive frames containing an unmatched detection before
           # a track is deleted

minHits = 1 # number of consecutive matches needed to establish a new track

ndThreshold = 0.0003    # if the cost of a 'match' between a detection and a tracker
                        # is below this - its not the same object and it needs its own
 tracker

minBoxArea = 40000      # Don't track boxes below this value - too small

# a list for tracker ids
trackerId = 1   # increment this to give an identity to new objects in the image stream
```

Trackers

Now, we set up our key data item - a list of trackers. At the moment its just a list - it'll become a list of `Tracker` objects whe we use it - defined by the `Tracker` class below.

In [4]:

```
# Main Object - our tracker list
trackers = [] # the tracker list
```

Tracker Class

Effectively, we need a something to keep track of trackers and it seems reasonable to associate the Kalman code with the tracker by incorporating the Kalman into a `Tracker` class.

The Kalman code is effectively unchanged from the `7_2_Kalman` lab but refactored into a class.

Each `Tracker` contains:

- `id` : its Identity
- `box` ; its bounding box
- `numHits` : how many frames its been detected in
- `numMisses` : how many frames since it was last detected
- and it's Kalman terms.

See `7_2_Kalman` for a description of the Kalman terms.

In [5]:

```
class Tracker():        # Class to keep track of trackers
    def __init__(self):
        # Initialise tracker's history
        self.id = 0                             # tracker's id
        self.box = np.zeros(shape=(2,2))        # bounding box co-ordinates
        self.numHits = 0                        # number of detection matches
        self.numMisses = 0                      # number of missed detections

        self.xState = np.matrix('0. 0. 0. 0.').T

        # Process matrix, assuming constant velocity model (x, y, x_dot, y_dot)
        self.F = np.matrix('''
                    1. 0. 1. 0.;
                    0. 1. 0. 1.;
                    0. 0. 1. 0.;
                    0. 0. 0. 1.
                    ''')

        # Measurement matrix, assumig we can only measure the co-ordinates
        self.H = np.matrix('''
                    1. 0. 0. 0.;
                    0. 1. 0. 0.
                    ''')

        # Initialise to all highly uncertain
        self.P = np.matrix(np.eye(4)*100)

        # Self motion -  we won't work the motion term in this example
        self.motion = np.matrix('0. 0. 0. 0.').T

        # Initialise the process covariance
        self.Q = np.matrix(np.eye(4))

        # Initialise the measurement covariance
        self.R = np.zeros(shape=(2,2))

    def kalmanFilter(self, box, R):
        # build z-term by getting box centres
        z = np.zeros(shape=(1,2))
        z[0][0] = (box[0][0] + box[1][0])/2
        z[0][1] = (box[0][1] + box[1][1])/2

        x = self.xState

        # Update step
        S = self.H*self.P*self.H.T + R
        K = self.P*self.H.T*S.I         # Kalman Gain
        y = np.matrix(z).T - self.H*x   # residual term
        x = x + K*y
        I = np.matrix(np.eye(self.F.shape[0]))
        self.P = (I - K*self.H)*self.P

        # Predict Step
        # Predict x and P based on measurement
        x = self.F*x + self.motion
        self.P = self.F*self.P*self.F.T +self.Q

        self.xState = x
```

```
    def box2xstate(self, box):
        # convert np.(2x2), [[x1, y1], [x2, y2]]
        # to state vector state_x [x, y, x_dot, y_dot]
        # by finding centre of box and using that as x,y
        self.xState[0] = (box[0][0] + box[1][0]) / 2 # centre x
        self.xState[1] = (box[0][1] + box[1][1]) / 2 # centre y

    def xstate2box(self):
        # use our  xState to update our box
        # by finding our box's centre, extracting
        # the new centre from xState and moving
        # our box by delta centres
        newCentre = np.zeros(shape=(2,1))
        newCentre[0] = self.xState[0]
        newCentre[1] = self.xState[1]
        oldCentre = np.zeros(shape=(2,1))
        oldCentre[0] = (self.box[0][0] + self.box[1][0]) / 2 # centre x
        oldCentre[1] = (self.box[0][1] + self.box[1][1]) / 2 # centre y

        return self.adjustBBox(self.box, oldCentre, newCentre)

    def adjustBBox(self, box, origCentre, newCentre):
        # Just move any box from oldCentre to newCentre
        delta = newCentre - origCentre
        adjustedBox = np.zeros(shape=(2,2))
        adjustedBox[0][0] = box[0][0] + delta[0]
        adjustedBox[0][1] = box[0][1] + delta[1]
        adjustedBox[1][0] = box[1][0] + delta[0]
        adjustedBox[1][1] = box[1][1] + delta[1]
        return adjustedBox
```

Helper Functions

getPngsAndBoxes

This is a helper function to get a list of png files in a directory and a file of bounding boxes for those pngs.

In [6]:

```
def getPngsAndBoxes():
    global pngDir
    global bbDir

    pngFolder = os.fsencode(pngDir)
    bbFolder = os.fsencode(bbDir)

    pngFiles = []
    for filename in os.listdir(pngFolder):
        if filename.decode().endswith(".png"):
            pngFiles.append(pngDir + filename.decode())
    pngFiles.sort()

    for filename in os.listdir(bbFolder):
        if filename.decode().endswith(".boxes"):
            bbFilename = bbDir + filename.decode()

    bbfh = open(bbFilename, "r")
    bbLines = bbfh.readlines()
    bbfh.close()

    return bbLines, pngFiles
```

Parse Detections

We'll use this function in the main loop to wrangle the detections into the format we want to supply to our Kalman Filter.

Essentially it takes the name of png file, an img object and the list of bounding boxes as inputs.

It then finds the correct record (if any) for that image in the bounding boxes list and converts the bounding box parameters into a format which we'll use for the rest of the program (it converts back to absolute pixel values).

It returns a centre and a confidence value for the image supplied to it.

In [7]:

```
#
# Helper Function
#
# Similar to Kalman example
#
# Takes an image and a set of yolo3 bounding boxes
# Finds all the bounding boxes above min box area
# in a frame and returns them as a list of centres, a list
# of bounding boxes and a list of probabiliy estimates
#
def parseDetections(bBoxes, pngFile, img):
    global minBoxArea

    index = int(re.findall(r'\d+', pngFile)[-1])

    imgH, imgW = img.shape[:2]

    centreList = []
    boxList = []
    confList = []

    for line in bBoxes:
        # ((x_plus_w+x)/2)/image.shape[1] # width
        # ((y_plus_h+y)/2)/image.shape[0] # height
        # (x_plus_w - x)/image.shape[1]
        # (y_plus_h - y)/image.shape[0]
        lineArray = np.genfromtxt(io.StringIO(line), delimiter=",")
        lineIndex = int(lineArray[0])
        if lineIndex == index:
            centre = np.zeros(shape=(2,1))
            box = np.zeros(shape=(2,2))
            conf = 0.000001 # hack to avoid div by zero
            centre = lineArray[2:4]
            halfW = lineArray[4] * imgW / 2
            halfH = lineArray[5] * imgH / 2
            conf += lineArray[6]
            centre[0] *= imgW
            centre[1] *= imgH
            box[0][0] = centre[0] - halfW   # x1
            box[0][1] = centre[1] - halfH   # y1
            box[1][0] = centre[0] + halfW   # x2
            box[1][1] = centre[1] + halfH   # y2
            boxW = halfW * 2
            boxH = halfH * 2
            boxArea = boxW * boxH
            if boxArea > minBoxArea:                # dump small boxes
                confList.append(conf)
                boxList.append(box)
                centreList.append(centre.tolist())

    return centreList, boxList, confList
```

drawBoxLabel

A helper function to draw a bounding box and put a label on it

In [8]:

```
#
# Helper Function
#
# Draw a box with a label.  Default label is 'untracked'
#
def drawBoxLabel(img, bbox, color=(0,255,255), label="Untracked"):
    font = cv2.FONT_HERSHEY_SIMPLEX
    fontSize = 1.2

    cv2.rectangle(img, (int(bbox[0][0]), int(bbox[0][1])), (int(bbox[1][0]), int(bbox[1
][1])), color, 8)
    cv2.putText(img, label, (int(bbox[0][0])-25,int(bbox[0][1])-25), font, fontSize, co
lor, 8, cv2.LINE_AA)
```

Hungarian Cost Term

This is very important term.

The Hungarian is effectively a cost minimisation function but we need to provide it with a way of expressing *the likelihood that two bounding boxes represent the same object* as a single number.

The way we're using the Hungarian function is it actually tries to find the max cost, so we want a term that is more **expensive** if the two boxes are likely to represent the same object and **cheaper** if the two boxes are not likely to represent the same image.

So, we need something that generates a larger cost if the boxes are likely to be the same object and a smaller cost if the boxes are unlikely to be the same object

The Hungarian will then find the set of relationships with the overall highest cost.

We simply used Euclidian distance - boxes that are closer together are more likely to be the same box. So, the smaller the distance between the centres of the boxes the more likely they are to represent the same object.

However, that's an inverse relationship to what our Hungarian needs so we simply invert it. Our best boxes have a difference of 0 between Kalman prediction and new detection so after inverting we can end up with a divide by zero so we just set it to 1 - the best possible match under our this scheme

This function could be a candidate for something like a linear regression:

- to understand contributions to a cost functon from things like
- Bounding Box Size
- Bounding Box Overlap
- Distance Bounding Boxes are from each other
- How long since we last saw an image
- How similar the images represented by the two boxes are. etc...

In [9]:

```
def boxCost(box1, box2):
    # width and height of box1
    # get centre of box1
    w1 = box1[1][0] - box1[0][0]
    w1 = w1/2
    cx1 = box1[0][0] + w1
    h1 = box1[1][1] - box1[0][1]
    h1 = h1/2
    cy1 = box1[0][1] + h1

    # width and height of box2
    # get centre of box2
    w2 = box2[1][0] - box2[0][0]
    w2 = w2/2
    cx2 = box2[0][0] + w2
    h2 = box2[1][1] - box2[0][1]
    h2 = h2/2
    cy2 = box2[0][1] + h2

    xDist = abs(cx2 - cx1)
    yDist = abs(cy2 - cy1)

    # square root of x squared plus y squared
    cost = xDist**2 + yDist**2
    cost = math.sqrt(cost)

    # Invert to get what we need for
    # this implementation of Hungarian
    if cost == 0:
        cost = 1
    else:
        cost = 1/cost   # bigger cost if closer

    return cost
```

assignDetectionsToTrackers

This operates by building a cost matrix of the current trackers and the new detections and using a Hungarian auction to re-order them for best cost.

It returns three lists:

- matches (matched)
- trackers without detections (unmatchedTrackers)
- detections without trackers (unmatchedDetections)

In [10]:

```
def assignDetectionsToTrackers(trackers, detections):
    global ndThreshold      # tweak the output of the Hungarian - it can produce poor m
atches

    # Build a cost matrix - all zeros (size determined by num trackers and detections
    # Set it up as float to match our cost function
    costMatrix = np.zeros((len(trackers), len(detections)), dtype=np.float32)

    # Fill the cost matrix with 'prices' derived from the
    # cost term
    # A cost for every combination of tracker and new detection
    for t,trk in enumerate(trackers):
        for d,det in enumerate(detections):
            costMatrix[t,d] = boxCost(trk,det)

    # Produce matches
    # Solve the maximising of the sum of cost asignment using the
    # Hungarian algorithm (aka Munkres algorithm)
    matchedRowIdx, matchedColIdx = linear_sum_assignment(-costMatrix)

    # First of all find any tracker that didn't find a date
    # with a new detection at all
    # add it to the unmatchedTrackers list
    # Maybe that object has gone away ...
    unmatchedTrackers, unmatchedDetections = [], []
    for t,trk in enumerate(trackers):
        if(t not in matchedRowIdx):
            unmatchedTrackers.append(t)

    # Now find any detection that didn't find a date
    # with an old trackeer  at all
    # add it to the unmatchedDetections list
    # Maybe its a new object
    for d, det in enumerate(detections):
        if(d not in matchedColIdx):
            unmatchedDetections.append(d)

    # Now, look at the matches in more detail
    # Maybe there's a few matches that are not
    # going to work
    matches = []

    # If the cost is than nd_theshold then
    # override the match - its not good enough
    # If you change the cost function, you'll probably
    # need to change ndThreshold as well
    for m, _ in enumerate(matchedRowIdx):
        if(costMatrix[matchedRowIdx[m],matchedColIdx[m]]<ndThreshold):
            # Nope, not really a match
            # add the detection to unmatched detections list
            # add the tracker to unmatched tracker list
            unmatchedTrackers.append(matchedRowIdx[m])
            unmatchedDetections.append(matchedColIdx[m])
        else:
            # Its a match
            # Record details of the match - tracker index and detection index
            match = np.empty((1,2),dtype=int)
            match[0][0] = matchedRowIdx[m]
            match[0][1] = matchedColIdx[m]
```

```
            # Add to matches list
            matches.append(match)

    # Clean and return
    if(len(matches)==0):
        matches = np.empty((0,2),dtype=int)
    else:
        matches = np.concatenate(matches,axis=0)

    return matches, np.array(unmatchedDetections), np.array(unmatchedTrackers)
```

demoHungarian

Here is the main code

For each png in a directory:

- open it
- get the bounding boxes for it
- attempt to match the bounding boxes with any existing trackers
- get three lists:
  - matches
  - untracked detections
  - trackers with no detections
- Handle all three cases
  - Update and display matches
  - Age (and prune) unmatched trackers
  - Create new trackers for unmatched detections

Finally pretty print matched detections and trackers to video.

In [11]:

```
def demoHungarian():
    global trackerId
    global writer
    global trackers

    # Initialise state to no position
    x = np.matrix('0. 0. 0. 0.').T
    # Initialise state uncertainty covariance
    P = np.matrix(np.eye(4))*100

    # Create an empty box
    box = np.zeros(shape=(2,2))

    # Get lists of files and bounding boxes
    bbLines, pngFiles = getPngsAndBoxes()

    # Main loop - do this for every image in the directory
    for pngFile in pngFiles:
        #print ("handling .." + os.path.basename(pngFile))
        # Load the file
        img = cv2.imread(pngFile)

        # Gather a list of new boxes and confidence values
        # We'll use this confidence in R term of Kalman
        # Derive R from yolo confidence level in detection
        _, newBoxes, newConfs = parseDetections(bbLines, pngFile, img)

        # Build our known boxes list by extracting it from our list
        # of tracker objects - each tracker has a box its minding for us
        knownBoxes = []

        if(len(trackers) > 0):
            for trk in trackers:
                knownBoxes.append(trk.box)

        # Now we have a list of old boxes being tracked and a
        # list of new boxes.
        # Hand over to assignment function to build our
        # three lists - matched, unmatched detections and unmatched trackers
        matched, unmatchedDetections, unmatchedTrackers \
        = assignDetectionsToTrackers(knownBoxes, newBoxes)

        # Deal with matched detections
        if(matched.size > 0):
            for trkIdx, detIdx in matched:
                # there was a match
                # new data for tracked object
                box = newBoxes[detIdx]
                conf = newConfs[detIdx]
                R = np.eye(2)
                R *= 1/conf
                # find tracker in list
                tmpTrk = trackers[trkIdx]
                # update its data and run a kalman filter
                tmpTrk.kalmanFilter(box, R)
                tmpTrk.box = box
                knownBoxes[trkIdx] = tmpTrk.box
                tmpTrk.numHits += 1
                tmpTrk.numMisses = 0
```

```
        # Deal with unmatched detections
        if (len(unmatchedDetections)>0):
            for idx in unmatchedDetections:
                box  = newBoxes[idx]
                tmpTrk = Tracker() # create a new tracker
                tmpTrk.box = box
                tmpTrk.box = tmpTrk.xstate2box()
                tmpTrk.id = trackerId # assign ID to tracker
                trackerId += 1
                trackers.append(tmpTrk)
                knownBoxes.append(tmpTrk.box)

        # Deal with unmatched tracks
        if (len(unmatchedTrackers)>0):
            for trkIdx in unmatchedTrackers:
                tmpTrk = trackers[trkIdx]
                tmpTrk.numMisses += 1
                tmpTrk.box = tmpTrk.xstate2box()
                knownBoxes[trkIdx] = tmpTrk.box

        # The list of tracks to be displayed
        for trk in trackers:
            if ((trk.numHits >= minHits) and (trk.numMisses <= maxAge)):
                drawBoxLabel(img, trk.box, label="Tracked " + str(trk.id))

        # clean up deleted tracks
        trackers = [x for x in trackers if x.numMisses <= maxAge]

        # Resize and show the image
        vidout = cv2.resize(img, (int(img.shape[1]/4), int(img.shape[0]/4)))

        # Build a frame of our output video
        if writer is None:
            # Initialize our video writer
            fourcc = cv2.VideoWriter_fourcc(*'VP80')
            writer = cv2.VideoWriter('video.webm', fourcc, 30, (vidout.shape[1], vidout
.shape[0]), True)

        # Write the output frame to disk
        writer.write(vidout)

    # Release the file pointers
    writer.release()

demoHungarian()

!ls video.webm
```

```
'ls' is not recognized as an internal or external command,
operable program or batch file.
```

Video

Thia code plays the video we just made.

The Tracked objects are boxed and labelled i yellow with an tracker number for each object.

In [12]:

```
# Set this to 1 if video display
# is not working - works with chrome and firefox, not with safari
videoBodge = 0

def arrayShow (imageArray):
    ret, png = cv2.imencode('.png', imageArray)
    encoded = base64.b64encode(png)
    return Image(data=encoded.decode('ascii'))

if(videoBodge == 0):
    from IPython.display import HTML
    from base64 import b64encode
    webm = open('video.webm','rb').read()
    data_url = "data:video/webm;base64," + b64encode(webm).decode()
else:
    video = cv2.VideoCapture("video.webm")
    while(video.isOpened()):
        clear_output(wait=True)
        ret, frame = video.read()
        if(ret == False):
          break
        lines, columns, _ =  frame.shape
        img = arrayShow(frame)
        display(img)
        time.sleep(1)
```

In [13]:

```
# Display Video
HTML("""
<video width=200 controls>
      <source src="%s" type="video/webm">
</video>
""" % data_url)
```

Out[13]:

Conclusion

Exercises

Exercise 1 Reduce the `minBoxArea` to effectively track smaller boxes - you may also need to assign a `colour` term to the tracker as I'd expect there to be more objects to track and it should become harder to see what the program is doing. What does happen?

Exercise 2 Rework `boxCost` to combine:

1. Euclidean Distance
2. One other term that you think might improve the cost function, e.g.:
    - similarity of bounding box size;
    - bounding box percentage overlap
    - etc...

Don't forget the `ndThreshold` term will need to be adjusted as well. Does it improve the tracking.

Optional Exercise 2 Rework `boxCost` to use Mahalanobis distance instead of Euclidean distance. See https://en.wikipedia.org/wiki/Mahalanobis_distance (https://en.wikipedia.org/wiki/Mahalanobis_distance) for details.

Takeaways

1. You've seen three items working in concert; a detector, a tracker and an allocator to track multiple objects in video.
2. You've seen that how you define your cost function in the allocation scheme is critical.
3. You've adjusted a simple cost function yourself.
4. You have the concept that you can use smarter trackers, smarter, detectors and a smarter allocation cost function if you have the computational budget for it.

Next Steps

1. We'll look at a completely different type of tracker now - based on Optical Flow
2. Finally, we'll review contemporary approaches to tracking using CNNs.