**MY CODE :-**

Original file is located at

<https://colab.research.google.com/drive/1mWlpQt5Dy9TYR4by2v6mNZDN3IKaCRp4>

"""

import cv2

import numpy as np

import json

from google.colab.patches import cv2\_imshow

class KalmanFilterColorDetector:

def \_\_init\_\_(self):

self.kalman = cv2.KalmanFilter(4, 2) # 4 states (x, y, vx, vy), 2 measurements (H, S)

self.kalman.measurementMatrix = np.array([[1, 0, 0, 0],

[0, 1, 0, 0]], np.float32)

self.kalman.transitionMatrix = np.array([[1, 0, 1, 0],

[0, 1, 0, 1],

[0, 0, 1, 0],

[0, 0, 0, 1]], np.float32)

self.kalman.processNoiseCov = np.array([[1e-4, 0, 0, 0],

[0, 1e-4, 0, 0],

[0, 0, 1e-4, 0],

[0, 0, 0, 1e-4]], np.float32)

def detect\_color(self, frame):

hsv = cv2.cvtColor(frame, cv2.COLOR\_BGR2HSV)

mask = cv2.inRange(hsv, (0, 70, 50), (10, 255, 255)) # Red color range

contours, \_ = cv2.findContours(mask, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

if len(contours) > 0:

contour = max(contours, key=cv2.contourArea)

x, y, w, h = cv2.boundingRect(contour)

center\_x = x + w // 2

center\_y = y + h // 2

measurement = np.array([[center\_x], [center\_y]], dtype=np.float32)

self.kalman.correct(measurement)

prediction = self.kalman.predict()

color = tuple(frame[center\_y, center\_x])

return int(prediction[0]), int(prediction[1]), color

return None

def run(self, video\_path, output\_json):

cap = cv2.VideoCapture(video\_path)

frame\_width = int(cap.get(cv2.CAP\_PROP\_FRAME\_WIDTH))

frame\_height = int(cap.get(cv2.CAP\_PROP\_FRAME\_HEIGHT))

fps = int(cap.get(cv2.CAP\_PROP\_FPS))

total\_frames = int(cap.get(cv2.CAP\_PROP\_FRAME\_COUNT))

results = []

while cap.isOpened():

ret, frame = cap.read()

if not ret:

break

result = {}

# Detect color and get object position

position = self.detect\_color(frame)

if position is not None:

result['frame\_index'] = int(cap.get(cv2.CAP\_PROP\_POS\_FRAMES))

result['position'] = [int(position[0]), int(position[1])]

result['color'] = [int(c) for c in position[2]] # Convert color to list of integer

results.append(result)

cv2\_imshow(frame) # Display frame using cv2\_imshow

if cv2.waitKey(1) & 0xFF == ord('q'):

break

cap.release()

cv2.destroyAllWindows()

# Save the collected information in a JSON file

with open(output\_json, 'w') as file:

json.dump(results, file)

print("Data saved to", output\_json)

if \_\_name\_\_ == '\_\_main\_\_':

detector = KalmanFilterColorDetector()

detector.run('video.mp4', 'output.json')

import cv2

import numpy as np

import json

from google.colab.patches import cv2\_imshow

class KalmanFilterColorDetector:

def \_\_init\_\_(self):

self.kalman = cv2.KalmanFilter(4, 2) # 4 states (x, y, vx, vy), 2 measurements (H, S)

self.kalman.measurementMatrix = np.array([[1, 0, 0, 0],

[0, 1, 0, 0]], np.float32)

self.kalman.transitionMatrix = np.array([[1, 0, 1, 0],

[0, 1, 0, 1],

[0, 0, 1, 0],

[0, 0, 0, 1]], np.float32)

self.kalman.processNoiseCov = np.array([[1e-4, 0, 0, 0],

[0, 1e-4, 0, 0],

[0, 0, 1e-4, 0],

[0, 0, 0, 1e-4]], np.float32)

def detect\_color(self, frame):

hsv = cv2.cvtColor(frame, cv2.COLOR\_BGR2HSV)

mask = cv2.inRange(hsv, (0, 70, 50), (10, 255, 255)) # Red color range

contours, \_ = cv2.findContours(mask, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

if len(contours) > 0:

contour = max(contours, key=cv2.contourArea)

x, y, w, h = cv2.boundingRect(contour)

center\_x = x + w // 2

center\_y = y + h // 2

measurement = np.array([[center\_x], [center\_y]], dtype=np.float32)

self.kalman.correct(measurement)

prediction = self.kalman.predict()

color = tuple(frame[center\_y, center\_x])

return int(prediction[0]), int(prediction[1]), color

return None

def run(self, input\_source, output\_json):

if input\_source == 'webcam':

cap = cv2.VideoCapture(0)

else:

cap = cv2.VideoCapture(input\_source)

results = []

while cap.isOpened():

ret, frame = cap.read()

if not ret:

break

result = {}

# Detect color and get object position

position = self.detect\_color(frame)

if position is not None:

result['position'] = [int(position[0]), int(position[1])]

result['color'] = [int(c) for c in position[2]] # Convert color to list of integers

# Store other relevant information

# ...

results.append(result)

cv2\_imshow(frame) # Display frame using cv2\_imshow

if cv2.waitKey(1) & 0xFF == ord('q'):

break

cap.release()

cv2.destroyAllWindows()

# Save the collected information in a JSON file

with open(output\_json, 'w') as file:

json.dump(results, file)

print("Data saved to", output\_json)

if \_\_name\_\_ == '\_\_main\_\_':

detector = KalmanFilterColorDetector()

detector.run('webcam', 'output.json')

import cv2

import numpy as np

import json

class KalmanFilterColorDetector:

def \_\_init\_\_(self):

self.kalman = cv2.KalmanFilter(4, 2) # 4 states (x, y, vx, vy), 2 measurements (H, S)

self.kalman.measurementMatrix = np.array([[1, 0, 0, 0],

[0, 1, 0, 0]], np.float32)

self.kalman.transitionMatrix = np.array([[1, 0, 1, 0],

[0, 1, 0, 1],

[0, 0, 1, 0],

[0, 0, 0, 1]], np.float32)

self.kalman.processNoiseCov = np.array([[1e-4, 0, 0, 0],

[0, 1e-4, 0, 0],

[0, 0, 1e-4, 0],

[0, 0, 0, 1e-4]], np.float32)

def detect\_color(self, frame):

hsv = cv2.cvtColor(frame, cv2.COLOR\_BGR2HSV)

mask = cv2.inRange(hsv, (0, 70, 50), (10, 255, 255)) # Red color range

contours, \_ = cv2.findContours(mask, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

if len(contours) > 0:

contour = max(contours, key=cv2.contourArea)

x, y, w, h = cv2.boundingRect(contour)

center\_x = x + w // 2

center\_y = y + h // 2

measurement = np.array([[center\_x], [center\_y]], dtype=np.float32)

self.kalman.correct(measurement)

prediction = self.kalman.predict()

color = tuple(frame[center\_y, center\_x])

return int(prediction[0]), int(prediction[1]), color

return None

def process\_image(self, image\_path):

frame = cv2.imread(image\_path)

if frame is None:

print("Failed to load the image:", image\_path)

return

result = {}

# Detect color and get object position

position = self.detect\_color(frame)

if position is not None:

result['position'] = [int(position[0]), int(position[1])]

result['color'] = [int(c) for c in position[2]] # Convert color to list of integers

# Store other relevant information

# ...

return result

def process\_images(self, image\_paths):

results = []

for image\_path in image\_paths:

result = self.process\_image(image\_path)

if result is not None:

results.append(result)

return results

def save\_results\_to\_json(self, results, output\_json):

with open(output\_json, 'w') as file:

json.dump(results, file)

print("Data saved to", output\_json)

if \_\_name\_\_ == '\_\_main\_\_':

detector = KalmanFilterColorDetector()

# Specify the paths of the input images

image\_paths = ['image1.jpg', 'image2.jpg', 'image3.jpg']

# Process the images and collect the results

results = detector.process\_images(image\_paths)

# Save the results to a JSON file

detector.save\_results\_to\_json(results, 'output.json')

—----------------------------------------------------------------------------------------------------------------------------

The Kalman filter is a widely used algorithm in signal processing and control systems to estimate the state of a dynamic system. Like in hyperparameter tuning, the Kalman filter has parameters that need to be set appropriately to achieve accurate state estimation.

Hyperparameters in the context of the Kalman filter could include parameters such as the process noise covariance, measurement noise covariance, and initial state estimate. Tuning these hyperparameters involves finding the optimal values for them, which leads to improved state estimation performance.

Similar to hyperparameter tuning techniques, different approaches can be used to tune the Kalman filter's hyperparameters. This can involve manual tuning based on domain knowledge or using more advanced techniques like grid search or Bayesian optimization, where the performance of the filter is evaluated with different hyperparameter values to find the best combination.

Model ensembling in the context of the Kalman filter can be seen as combining multiple Kalman filters to enhance state estimation. By running multiple Kalman filters with different initial conditions or noise assumptions, and combining their estimated states using techniques like weighted averaging or voting, we can achieve more accurate and robust state estimates.

In summary, just as hyperparameter tuning aims to find the best parameter values for a machine learning model, tuning the hyperparameters of a Kalman filter helps optimize its state estimation performance. Similarly, model ensembling with the Kalman filter involves combining multiple filters to improve the accuracy and reliability of state estimates