## 💬 **تابع do\_synopsis**

این تابع تیوب‌هایی که در مرحله‌ی قبل استخراج شده‌اند را به هم می‌چسباند و خلاصه می‌کند. این تابع tube\_list, out, label\_flag, overlap\_ignoring را به عنوان ورودی می‌گیرد. پارامتر tube\_list یک لیست از تمام تیوب‌هایی است که قرار است آن‌ها را سیناپس کنیم. پارامتر label\_flag مشخص می‌کند که کنار آبجکت‌ها لیبل قرار بگیرد یا نه (رجوع شود به توضیحات setting.json). پارامتر out آبجکتی است که خروجی روی آن نوشته می‌شود (این آبجکت در فایل main تولید شده است). از پارامتر overlap\_ignoring برای بررسی تداخل دو تیوب با یکدیگر استفاده میشود. در اصل این پارامتر به ما نشان میدهد که تا چه میزان از تداخل دو تیوب را میتوانیم نادیده بگیریم.

ابتدا لیست tube\_list به ترتیب به سه تابع detection\_merge و eliminate\_consistent\_tubes و tubes\_concatenation داده می‌شود که این توابع تیوب‌ها را سیناپس می‌کنند. سپس خروجی سیناپس شده در فایل out نوشته می‌شود که در ادامه این پروسه توضیح داده می‌شود.

در سیستم تیوب‌ها دو نوع شماره‌ی فریم وجود دارد. یکی شماره‌ی مطلق فریم که نشان می‌دهد این فریم متناظر کدام فریم از ویدیوی ورودی است و یکی شماره‌ی فریم نسبی (shift) که بعد از سیناپس کردن به وجود می‌آید و نشان می‌دهد این فریم متناظر کدام فریم خروجی است.

| max\_frame\_shift = do\_max\_frame\_shift(tube\_list) |
| --- |

تابع do\_max\_frame\_shift یک لیست از تیوب‌ها را می‌گیرد و بیشترین شماره‌ی نسبی را محاسبه می‌کند. معنی max\_frame\_shift این است که ویدیوی‌ خروجی حداکثر چند فریم خواهد بود. بنابراین با معلوم بودن حداکثر شماره فریم‌ها در خروجی، می‌توانیم متغیر frame\_shift\_number را از ۰ تا max\_frame\_shift سوئیپ کنیم، آبجکت‌هایی که قرار است در آن فریم باشد را انتخاب کنیم و روی بک‌گراند اضافه کنیم و در نهایت نتیجه را در متغیر out بنویسیم. اینکار در یک حلقه‌ی while انجام می‌شود که در ادامه داخل این حلقه توضیح داده می‌شود:

به ازای هر فریم یک باز بک 'background.jpg' خوانده می‌شود و در متغیر background ریخته می‌شود.

| background = cv2.imread('background.jpg') |
| --- |

پارامتر flag\_write نشان می‌دهد که آیا خروجی حلقه در ویدیوی خروجی نوشته شود یا نه. در ابتدا این پارامتر مقدار ۰ دارد در صورتی که خروجی بدون مشکل تولید شود این مقدار ۱ می‌شود.

به ازای هر frame\_shift\_number (هر فریم خروجی) یک بار تمام تیوب‌های موجود tube\_list بررسی می‌شوند. شرط

| if (tube.f\_start\_shift <= frame\_shift\_number) and (tube.f\_end\_shift >= frame\_shift\_number) |
| --- |

مشخص می‌کند frame\_shift\_number بین f\_start\_shift و f\_end\_shift (شماره‌ی فریم نسبی ابتدا و انتهای تیوب) هست یا نه. اگر این شرط برقرار بود یعنی محتوای این تیوب (به طور خاص آن فریمی از تیوب که شماره نسبی ان frame\_shift\_number است) باید در خروجی قرار بگیرد. همانطور که قبلا گفته شده به منظور کاهش فضای اشغال شده در رم، ماشین‌های کراپ شده هر تیوب روی هارد ذخیره شده‌اند و باید به ازای هر باکس، تصویر کراپ شده لود شود

| if (not tube.cropped) and tube.boxes:  print("crop loading...")  tube.cropped, tube.motion = load\_dump(tube) |
| --- |

کد بالا تیوب tube را به تابع load\_dump تحویل می‌دهد و لیست کراپ‌های tube.cropped را که متناظر متناظر با لیست tube.box است را لود می‌کند. این کار یک ‌مرتبه یعنی اولین باری که در حلقه اصلی یک تیوب در حال بررسی است انجام می‌شود ولی در سایر دفعاتی لزومی نیست. دقت شود که با آپدیت شدن مقادیر متغیر tube مقادیر در tube\_list هم خودکار آپدیت می‌شوند.

| if (not tube.cropped) and tube.boxes:  print("crop loading...")  tube.cropped, tube.motion = load\_dump(tube) |
| --- |

در این مرحله پارامتر tube حاوی تمام اطلاعات مورد نیاز است. بنابراین این اطلاعات شامل کراپ و باکس، یکی یکی استخراج می‌شود

| bbox = tube.boxes.pop(0)  cropped\_image = tube.cropped.pop(0)  frame\_motion = tube.motion.pop(0) |
| --- |

در ادامه تصویر کراپ شده، به همراه مختصات باکس و طول و عرض فریم به تابع do\_mask و do\_bgremove داده می‌شود که خروجی آن تصویری است که در مکان باکس قسمت کراپ شده قرار گرفته و بک‌گراند آن نیز حذف شده است



حالا برای اینکه این تصویر را بک‌گراند ادغام کنیم ابتدا یک ماسک از آن ایجاد می‌کنیم:

| success, background\_mask = cv2.threshold(  masked\_image, 1, 255, cv2.THRESH\_BINARY\_INV) |
| --- |

## 

آنگاه این ماسک را با بکگراند ادغام می‌کنیم

| background\_mask\_grayscale = cv2.cvtColor( background\_mask, cv2.COLOR\_BGR2GRAY)  masked\_background = cv2.bitwise\_and( background, background, mask=background\_mask\_grayscale) |
| --- |



D

همچنین برخی از تیوب ها شامل باکس هایی هستند که با باکسهای سایر تیوبها مقداری تداخل دارند. این باکسها در لیست alpha\_decrease هر تیوب ذخیره شده است. اگر در ساختن فریم نهایی با استفاده از تیوب ها به دست آمده و پردازش شده نوبت به باکسی برسد که این شرایط را دارد، مقداری crop مربوط شفاف سازی (transparent) میشود و سپس در فریم نهایی قرار می گیرد. همان طور که در تکه کد زیر مشاهده می کنیم، اگر باکس جز باکس هایی باشد که شامل فرایند شفاف سازی می گردد، با شفافیت حدود 60 درصد در فریم نهایی قرار می گیرد، در غیر این صورت بدون تغییر به فریم نهایی اضافه می گردد.

| if frame\_shift\_number in set(tube.alpha\_decrease):  temp = cv2.add(masked\_image, masked\_background, dtype=cv2.CV\_8U)  background = cv2.addWeighted(temp, .6, background, 0.4, 0)  else:  background = cv2.add(masked\_image, masked\_background, dtype=cv2.CV\_8U) |
| --- |

اگر label\_flag برابر ۱ بود شماره آغاز تیوب و اندیس تیوب کنار آبجکت‌های آن نوشته می‌شود. در صورتی که به این مرحله رسیدیم یعنی یک خروجی معتبر داریم بنابراین flag\_write برابر یک می‌شود . این لوپ برای تیوب‌های مختلف تکرار می‌شود و تمام آبجکت‌های آن‌ها به این فریم اضافه می‌شود. در آخر اگر flag\_write برابر یک بود یعنی حداقل یک آبجکت در آن فریم وجود دارد و فریم به ویدیوی خروجی اضافه می‌شود.

## 💬 **تابع do\_crop**

این تابع یک تصویر و مختصات یک باکس را دریافت می‌کند و آن باکس را کراپ می‌کند.

## 💬 **تابع do\_mask**

این تابع یک تصویر کراپ شده‌ی cropped\_image و مختصات آن در فریم یعنی bbox را و همچنین H و W ویدیوی ورودی را دریافت می‌کند و یک ماسک بر می‌گرداند که همه‌ جای آن به غیر از محل تصویر کراپ شده سیاه است.



## 💬 **تابع do\_max\_frame\_shift**

این تابع یک tube\_list را دریافت می‌کند و شماره‌ی نسبی f\_end\_shift و f\_start\_list در بین تیوب‌های آن را محاسبه می‌کند

| for tube in tube\_list:  f\_start\_list.append(tube.f\_start\_shift)  f\_end\_list.append(tube.f\_end\_shift) |
| --- |

آنگاه بیشینیه f\_end\_shift و کمینه f\_start\_list را از هم کم می‌کند. از آنجا که کمینه f\_start\_list همیشه ۰ است در واقع این تابع همیشه بیشینه‌ی f\_end\_shift را می‌دهد که مفهوم آن بیشترین تعداد فریم در ویدیوی خروجی متناظر با آن لیست از است

## 💬 **تابع detection\_merge**

وقتی کار تشخیص و ردیابی اشیا در یک بافر تمام شد، انبوهی از تیوب های ساخته شده موجود میباشند که هرکدام در اثر تشخیص یک شی و ردیابی آن در 30 فریم بعدی، تشکیل شده اند. این تیوب ها هر کدام نماینده یک شی در ویدیو می باشند ولی یک سری از اشیا هستند که در تعداد detection هایی بیشتر از یکی رویت شده اند ولی اطلاعات این اشیا در تعدادی از تیوب های مختلف جدا از هم پخش شده است. لذا لازم است تا تیوب های مربوط به یک شی به هم وصل شوند و یک تیوب واحد را تشکیل دهند. این تابع در اصل این کار را انجام می دهد. برای تشخیص این که چه تیوب هایی باید به همدیگر متصل شوند از دو معیار کمک گرفته شده است.

معیار اول این است که دو تیوب در دو detection پشت سر هم آمده باشند. بدین معناست که اگر تیوب اول در detection چهارم آمده است ، تیوب دیگر باید یا در detection سوم و یا در detection پنجم آمده باشد. در غیر این صورت این دو تیوب شرط لازم را برای به هم وصل شدن ندارند. برای چک کردن این شرط از id های اختصاص داده شده به تیوبها استفاده میگردد. در اصل شروط cond 1 و cond 3 برای چک کردن این حالتها مورد استفاده قرار میگیرند.

| id\_1 = tube\_list[i].id[1] # detection ID  id\_2 = tube\_list[j].id[1] # detection ID    cond1 = (id\_2[0] == id\_1[-1] + 1) # tube2 comes after tube1  cond3 = (id\_2[-1] == id\_1[0] - 1) # tube1 comes after tube2 |
| --- |

معیار دوم برای بررسی شرایط وصل شدن دو تیوب میزان مساحت مشترک باکس انتهایی تیوب جلوتر(از لحاظ زمانی) و باکس ابتدایی تیوب عقب تر(از لحاظ زمانی)، می باشد. شروط cond 2 و cond 4 برای این منظور مورد استفاده قرار میگیرند.

| box\_1\_start = tube\_list[i].boxes[0]  box\_1\_end = tube\_list[i].boxes[-1]  box\_2\_start = tube\_list[j].boxes[0]  box\_2\_end = tube\_list[j].boxes[-1]  cond2 = do\_area(box\_1\_end, box\_2\_start) > 0.5  cond4 = do\_area(box\_2\_end, box\_1\_start) > 0.5 |
| --- |

حال اگر همزمان شروط cond1 و cond 2 برقرار باشند و یا همزمان شروط cond3 و cond4 برقرار باشند، فرض بر این می شود که دو تیوب مربوط به یک شی یکسان هستند و لذا فرایند متصل سازی آن ها انجام میپذیرد. برای انجام این کار نیز ابتدا اطلاعات ذخیره شده هر کدام از تیوب ها در pickle ها توسط تابع load\_dump مجددا بازیابی می شود، سپس اطلاعات مربوط به crop ها و motion ها با هم append میگردند و در نهایت تیوب های قبلی حذف میشوند و تیوب جدید ساخته شده که شامل اطلاعات دو تیوب قبلی هست جایگزین میگردد. سپس دوباره اطلاعات حجیم تیوب جدید ساخته شده dump میگردد.

## 💬 **تابع eliminate\_consistent\_tubes**

گاهی اوقات اشیایی در طول ویدیو به طور کامل ثابت هستند و حرکتی ندارند (مانند یک ماشین پارک شده). این اشیا در مرحله های detection مدام شناسایی می شوند و فرایند ردیابی روی آنها صورت می پذیرد. حضور ابن اشیا در ویدیوی نهایی ساخته شده باعث افزایش بیهوده طول ویدیو میشود، در حالی که این اشیا حاوی هیچ اطلاعات خاصی نیستند. لذا هدف این تابع، حذف اشیا ثابت از ویدیوی نهایی می باشد. برای تشخیص اشیا ثابت از تاریخچه سرعت آن شی در detection های متوالی استفاده شده است. به طوری که اگر در یک detection فقط از عدد فرض شده 20 بیشتر باشد، آن شی متحرک در نظر گرفته میشود. لازم به ذکر است که اعداد در نظر گرفته شده برای سرعت یک تیوب فاصله اقلیدسی بین باکس اول و باکس آخر آن تیوب در 30 فریم متوالی می باشد.

## 💬 **تابع do\_area**

از این تابع برای بررسی میزان تداخل دو باکس با هم، استفاده می گردد.

## 💬 **تابع do\_dump**

در حین انجام مراحل ساخت تیوبها برخی از فیلدها همچون crop و motion از لحاظ میزان حافظه ای که اشغال می کنند حجیم به حساب می آیند. لذا با پردازش فریمهای متوالی حجم حافظه ی اشغال شده به شدت افزایش می یابد و بعد از مدت کوتاهی حافظه سیستم پر میشود و امکان ادامه فرایند و پردازش فریمهای بیشتر در یک بافر وجو نخواهد داشت. لذا مجبور خواهیم بود تا سایز بافر را کاهش دهیم تا تمامی داده های حجیم موجود در تیوبها در حافظه فضا داشته باشند. برای بهبود این مسئله و امکان افزایش سایز بافر از ایده ی dump کردن اطلاعات حجیم همچون crop و motion استفاده گردیده است. زمانی که یک مرحله فرایند detection و tracking انجام گردید، تعدادی تیوب با توجه با اشیا موجود ویدیو ساخته خواهد شد. این تیوب ها برای انجام فرایند dump ارسال خواهند شد.

## 💬 **تابع load\_dump**

این تابع برای لود کردن مجدد اطلاعات pickle شده مورد استفاده قرار می گیرد. در طول اجرای برنامه در دو جا از این تابع استفاده شده است.

یکی در تابع detection\_merge که تیوب ها ساخته شده اند و قصد داریم تیوب های مربوط به یک شی را به هم بچسبانیم از تحت شرایط خاصی که توالی تیوب ها برقرار باشد و باکس انتهایی تیوب جلوتر(از لحاظ زمانی) و باکس ابتدایی تیوب عقب تر(از لحاظ زمانی) با هم در حدود 50 درصد تداخل داشته باشند، تیوب ها را مجددا لود میکنیم.

| cropped1, motion1 = load\_dump(tube\_list[i])  cropped2, motion2 = load\_dump(tube\_list[j]) |
| --- |

و اطلاعات آنها را با هم یکی می کنیم. یعنی crop ها و motion ها با هم append میگردند و مجددا فرایند do\_dump برای تیوب جدید که شامل دو تیوب قبلی است انجام می پذیرد.

دوم در حالتی که میخواهیم ویدیو نهایی را با استفاده از تیوب های به دست آمده و پردازش شده، بسازیم. یعنی در تابع do\_synopsis تیوب ها لود می گردند و از motion ها و crop ها برای جایگذاری مناسب شی در فریمهای ویدیوی نهایی بهره برده میشود.

## 💬 **تابع delete\_dump**

این تابع برای حذف dump ساخته شده به کار می رود. در اصل یک آرگومان که یکی از تیوب های ساخته شده میباشد به این متد پاس داده می شود و این تابع فایل pickle مربوط به آن تیوب را حذف می نماید. این تابع در زمانی که می خواهیم تیوب های مربوط به یک شی را به هم بچسبانیم در تابع detection\_merge به کار میرود. در اصل تیوب های قدیمی با این تابع از حافظه پاک می شوند و به جای آن تیوب جدیدی که شامل تیوب های قبل هست، جایگزین می گردد.

## 💬 **تابع do\_fix\_multi\_box**

در هنگام فرایند detection توسط yolo گاهی برخی از اشیا دو بار توسط این شبکه شناسایی می گردند. یعنی در بعضی مواقع در یک فریم مشخص دو باکس توسط yolo تشخیص داده می شوند که هر دو متعلق به یک آبجکت می باشند. این باکس ها معمولا هر کدام ناقص هستند و به تنهایی تصویر خوبی از کل آبجکت به ما نمی دهند. وظیفه تشخیص موارد این چنینی و اجتماع گرفتن از باکسها یک شی یکسان می باشد تا باکس حاصله شامل اطلاعات کاملتری از آبجکت باشد.