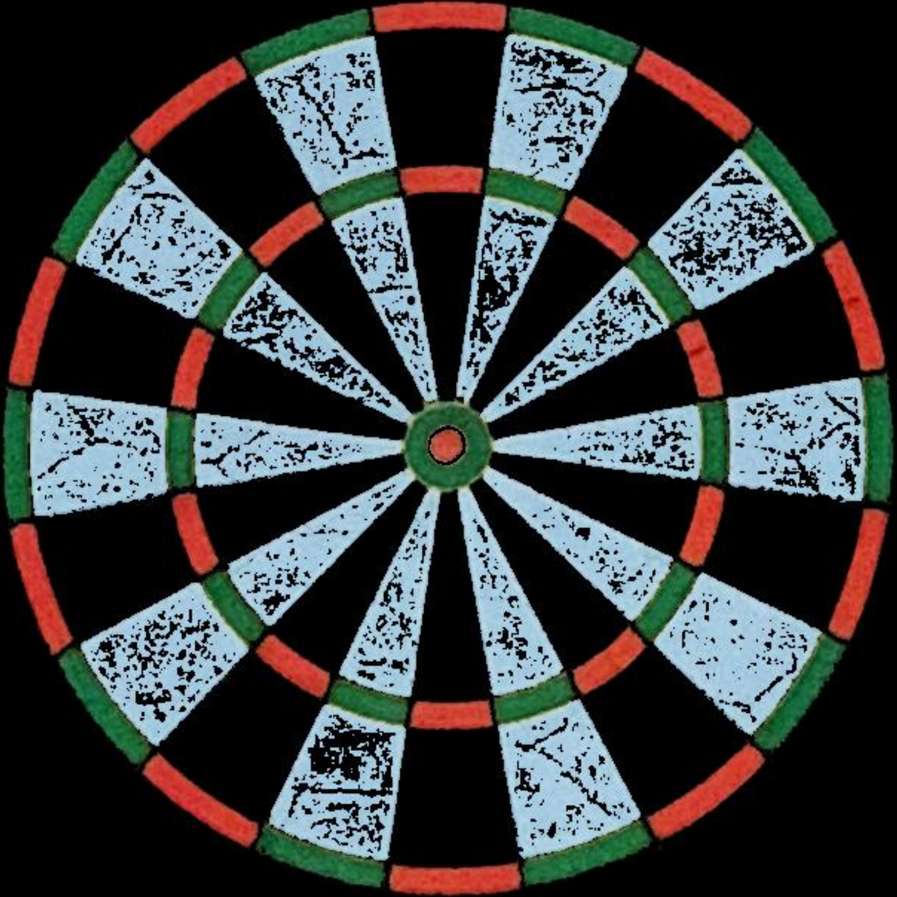


DARTs!

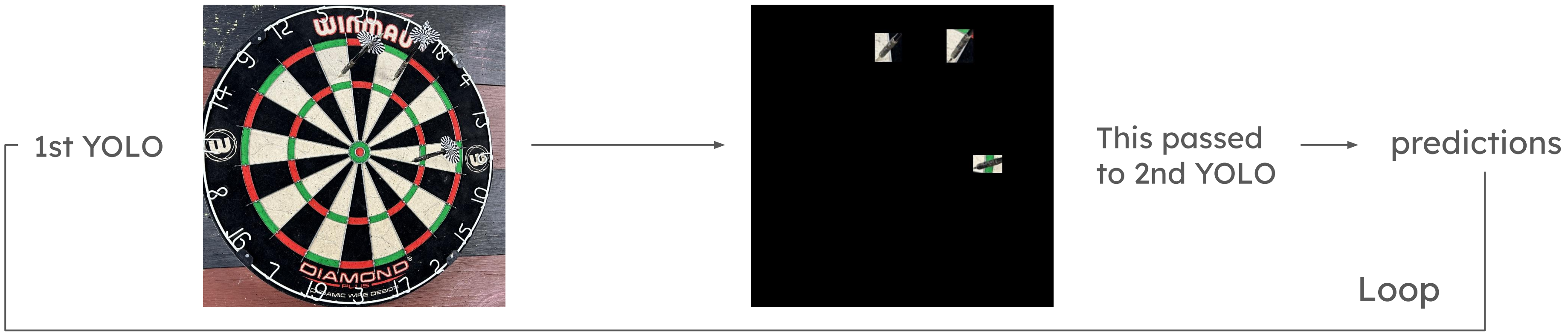
star me!

How does this work?
Let me explain you main points real quick

Starting from “raw” picture:

1. First YOLO model. Trained to detect 2 classes - the number of darts on the board and their very approximate position. Only 130 images were used to train first yolo, acceptable accuracy though. Crops image around darts with some initial padding, leaving other space on picture blacked out

2. Second YOLO model. Trained to detect one class - spot were dart hits the board. Takes the image from the first yolo and then, if number of predicted spots NOT the same as number of predicted darts, first YOLO increase/decrease padding, crops and passes picture back and all over again, until the number of predicted points and actual darts matches.



Data gathering: 1k images taken

130 images labeled for the 1st YOLO (2 classes)
1k images labeled for the 2nd YOLO (1 class, coordinates as center of bounding box)

What’s next?

YOLO the 2nd outputs the coordinates, let’s say “predictions”. But these points should be recalculated to actual score. And that’s way more tricky than it seems. Basically, idea is to calculate angle and distance from the center to each prediction, and since all dart boards are standard-sized circles, we can calculate a score.
BUT
When you’re looking at the circle-shaped board from an angle it’s looks like an ellipse! Even small angles could cause wrong score, especially when dart hits right in the edge between sectors. So the only way is to transform the board into the perfectly cropped circle.
As better as possible.

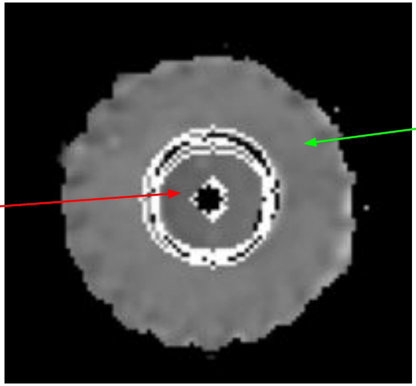
So, the most difficult part is:

3. **Transform perspective**, along with predictions, because any crops/warpTransforms cause the predictions shifts.

Let’s break out main steps of image perspective transformation!

Here we go!

1. Obviously, you need to detect center. Actually, center detection is crucial for correct transformation and further scoring, and it is hard used numerous times during transformation. Method crops radially small part near actual center of the image, then green and red mask applied.
   (Note: center, also known as bullseye, always red and surrounded by green circle on dart board)
   After that, dbscan clusters those green and red zones into the one thing, this prevents of unwanted points around the bullseye. Again, bulls eye not really needs to be in actual center of image, so cropping isn't perfect at all. When dbscan is done, I use HoughCircles function from OpenCV library. It tries to draw any possible circles through points present on the image. The center will be the average centers of all circles found. HoughCircles is really messy, can draw a bunch of false circles, but i need only around the bullseye, which actually ensured by dbscan. If none circles found, method returns to initial state, moves the point around which it cropped initially and crops image again.
   Thus, method searches for a center iteratively in some area!



This is a red part, inner bullseye

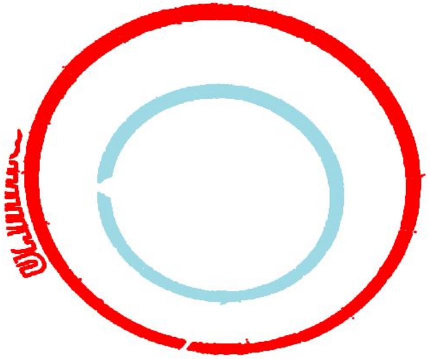
This is green, so called outer bullseye

Okay, I got a center

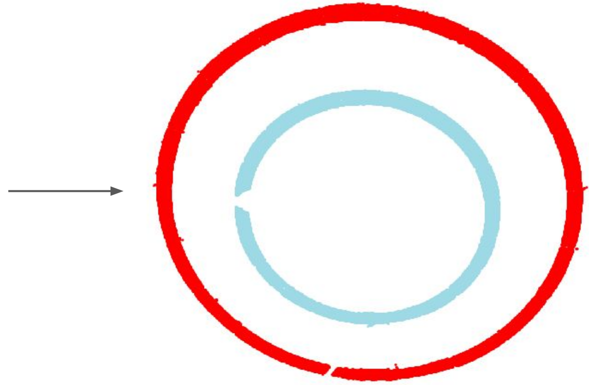
2. Try to fit ellipses around two dart board circles, which are double and triple zones. They both always red and green, so i now apply a green/red mask, apply dbscan and take 2 largest clusters! But what if some stuff around the board is red and green? To ensure that stuff will not be passed to dbscan, i simply crop around the center found, with some typical size. It should be perfect at all, just should cut the main artifacts around. Why i need dbscan, if i can just use red/green mask? Because *I need to find coordinates* of these circles, and dbscan labels the points on the picture! These points are wide spreaded, so I turn picture around the center and each time extract maximums of (x,y) coordinates from that mess. That provides points, which are almost perfectly fit around the both circles! But again, that's not a circles. Further, we will transform them to circles, but not yet. Next step is to fit around that points known ellipses, with OpenCV.
More then, this whole method iteratively loops through the dbscan parameters until both fitted ellipses have same semiaxis ratio, also known as eccentricity!

Here is how it looks like!

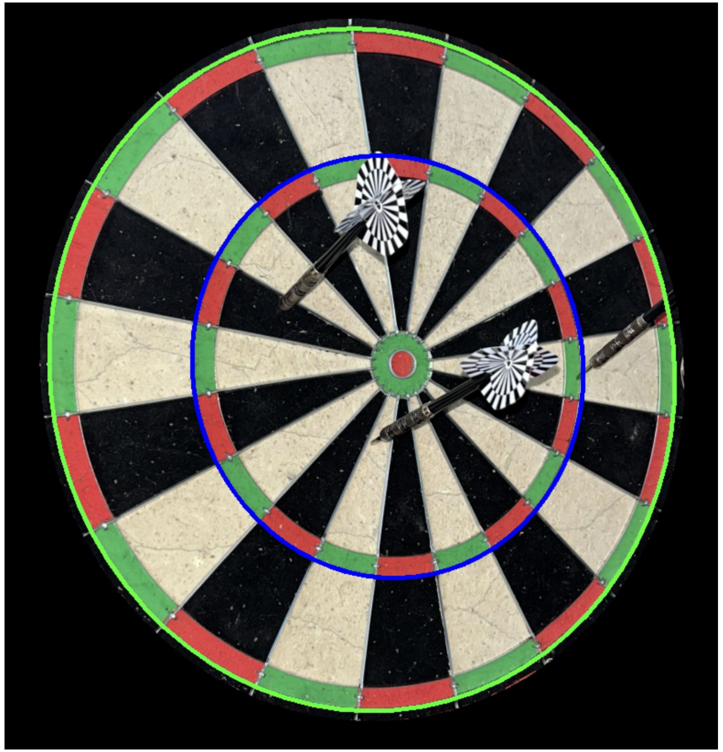
Bad dbscan, artifacts present, proceed to looping through eps and min_samples params



Looks better!

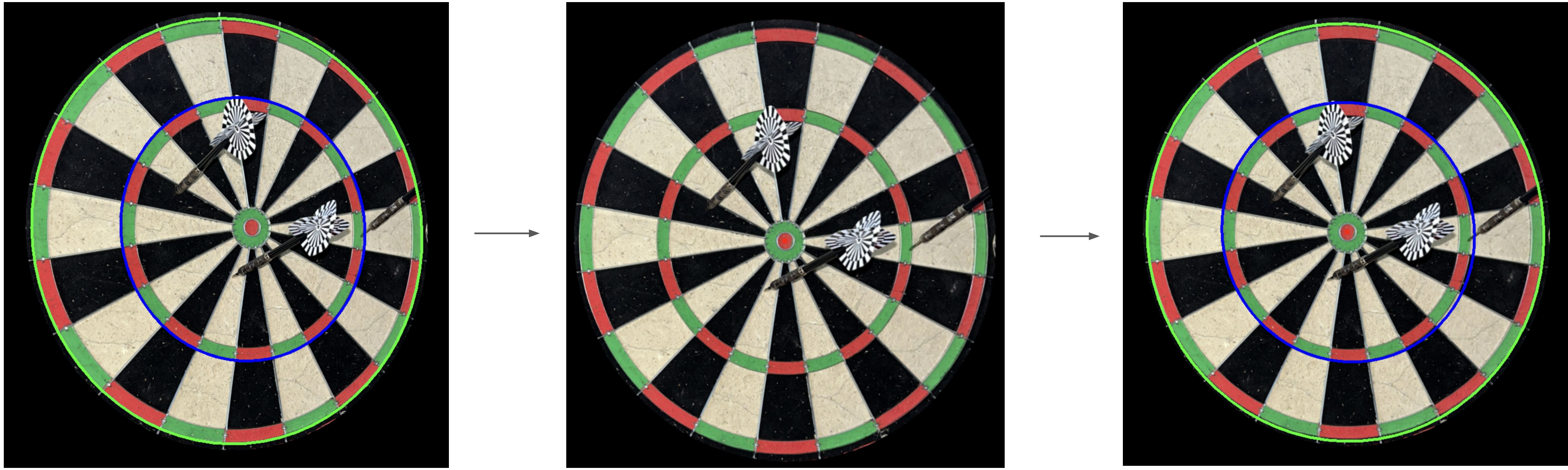


Final result:



Now transform!

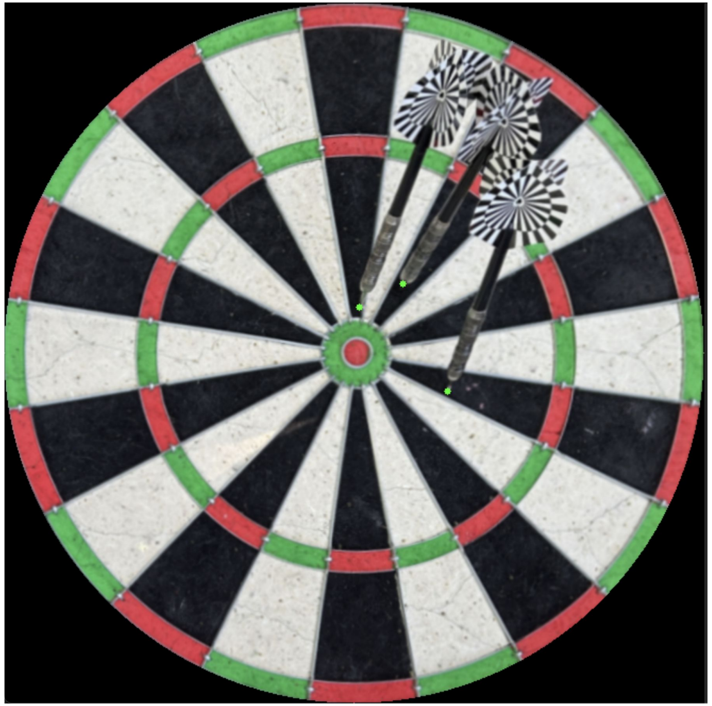
Repeat that until some accuracy is achieved



I just mentioned main ideas, a lot is left behind the scenes

Let’s score! It’s pretty easy now

Final transform:
Green points are predictions, that were transformed along with picture!



$\longrightarrow$