# Tracking Objects using QR codes and Deep Learning

Ali Ahmadinia[1]*, Atika Singh[1], Kambiz Hamadani[2], Yuanyuan Jiang[1]

[1] Department of Computer Science and Information Systems
California State University San Marcos
*aahmadinia@csusm.edu

[2] Department of Chemistry and Biochemistry
California State University San Marcos

## ABSTRACT

Despite recent advances in deep learning, object detection and tracking still require considerable manual and computational effort. First, we need to collect and create a database of hundreds or thousands of images of the target objects. Next we must annotate or curate the images to indicate the presence and position of the target objects within those images. Finally, we must train a CNN (convolution neural network) model to detect and locate the target objects in new images. This training is usually computationally intensive, consists of thousands of epochs, and can take tens of hours for each target object. Even after the model training in completed, there is still a chance of failure if the real-time tracking and object detection phases lack sufficient accuracy, precision, and/or speed for many important applications. Here we present a system and approach which minimizes the computational expense of the various steps in the training and real-time tracking process outlined above of for applications in the development of mixed-reality science laboratory experiences by using non-intrusive object-encoding 2D QR codes that are mounted directly onto the surfaces of the lab tools to be tracked. This system can start detecting and tracking it immediately and eliminates the laborious process of acquiring and annotating a new training dataset for every new lab tool to be tracked.

**Keywords:** Deep Learning, Object Detection and Tracking, QR code

## 1. INTRODUCTION

The current methods of object tracking require computationally intensive training on GPUs and huge datasets, including hundreds or even thousands of annotated images. We have created a system that can reduce or in some cases, eliminate these steps and make object tracking quicker and less expensive. This system can detect and track objects by only using QR codes. To achieve this, we first trained YOLOv4 to detect QR codes. For detection of new objects, we do not need to train the model again; all we have to do is generate a QR code that encodes the object's information (ie. its identity, height, and width) and mount this QR code on the a surface of the object. The system will then start detecting and tracking the object in real-time. The target application of this project is a virtual laboratory that requires detection and tracking of laboratory tools in real time. The biggest challenge for creating such a virtual environment is to be able to continuously add new tools. The traditional way to accomplish this would be to train deep learning models for each and every tool, which is very expensive and time-consuming. The greatest advantage of this project is that it circumvents the need to train deep learning models. Instead, we can train a model just once to detect and estimate the position of the QR codes, and then add any new tools into then system by printing out a calibrated QR code and attaching it to the object. This saves human and computational resources and provides an economical way to scale the number of optically-tracked lab tools for virtual/mixed reality science lab applications.

## 2. RELATED WORK

Numerous studies have employed QR codes that are detected and localized using deep learning models under natural settings. This project presents a prototype that demonstrates simplified object detection using a deep learning model which minimizes the resource intensive steps of data set preparation and training.

A recent evaluation of deep learning techniques for QR code detection [6] assessed the fidelity of object detection in the context of different deep learning model configurations; the authors proposed various modifications of the basic

architecture to take advantage of object subpart annotation. These modifications were implemented in the best scoring model and compared with a traditional technique. which demonstrated substantial improvements in object detection fidelity. In Recognition of Perspective Distorted QR Codes with a Partially Damaged Finder Pattern in Real Scene Images [1], the authors created a simple sequential algorithm that can localize a QR code image. It first implements image binarization, then localizes the QR code by utilizing a characteristic finder pattern located in the three corners of the target QR code. Finally, it identifies and evaluates perspective disorders. Their algorithm was able to recognize damaged finder patterns, is efficient, and works well for low-resolution images and camera systems. In Fast QR Code Detection in Arbitrarily Acquired Images[2], a 2-step process was employed to detect QR codes in arbitrary images. They use the Viola-Jones rapid object detection framework to focus the object detection algorithm on promising regions of the image. This study carried out extensive optimization of the parameters of the framework in order to improve finder pattern detection. In the second step, the algorithm uses the results of the first step to decide if the detected finder patterns belong to a particular QR code.

## 3. DESIGN AND IMPLEMENTATION

In the present work, we trained the YOLOv4 object detection model to detect QR codes. We used YOLOv4 because it outperformed other 2-stage CNN models such as Faster R-CNN in terms of accuracy and precision. YOLOv4 has also outperformed other 1-stage or one-shot object detection algorithms. To analyze this, we trained YOLOv4 using a publicly available dataset. This training used the transfer learning technique where we trained our model over existing weights resulting from previous training of YOLO on MS COCO dataset. The results were compared with the publicly available dataset in terms of mean average precision. Our analysis shows that YOLOv4 is approximately 30% better than MobileNet, 40% better than VGG16 and RestNet50. The results clearly show that YOLOv4 outperforms other single-shot object detectors like MobileNet, VG16 and ResNet50. For this reason, we chose YOLOv4 for our project.

After enabling QR code detection, we created algorithms that detect a QR code and decode it using PyzBar; we have also optimized this decoding process to increase throughput in terms of frames per second. The outputs from inference and decoding are processed using our custom algorithms to scale detections of a QR code to track a completely new associated object in real time. This includes optimized decoding, coordinate sorting, calibrated scaling, and continuation of detections using the region of interest.

### 3.1 Design Flow Phases

The first phase was to detect a QR code using YOLOv4[15] and then decode the QR code. When we used open-source decoders for the whole task, the decoding throughput in terms of frames per second was slower. To avoid performance degradation, we trained YOLOv4 model for QR codes. To increase accuracy, we used a publicly available dataset [6] and trained our model on top of weights from our previous training. Once YOLOv4 was trained, we used it for detecting QR codes. We fed the output of detections to the open-source PyzBar[3] QR code decoder. This improved the frames per second rate of QR decoding as the decoder did not have to process the entire image.

The next phase was to extrapolate this decoded information to start tracking a completely new object. To make this possible, the information required about the object was its name and its size (width and height), some objects can have multiple faces and each face can have a different height or width. One way to do this was to just encode the object's name in the QR code and rest of the information could be included in the source code. This method would have worked but scaling this project to add new objects would have been difficult. The source code had to be understood by the user and changed for every new object. To overcome this difficulty, we encoded the name of the object, object's width and height with respect to QR code's width and height. The height and width will be used for scaling the coordinates of QR code to include the object. Now, to track a new object using our model, one must calibrate its QR code and paste it on the object. After detecting and decoding the QR code, the output includes object's name, height and width scaling multipliers and the QR code's coordinates. The problem we faced was that the coordinates were not in any ordered form. They are jumbled up and it is difficult to figure out which vertex is which. To figure out the correct coordinates we came up with an algorithm that will sort them based on the x-coordinate values. We came up with a concept that the coordinate with maximum x-value and coordinate with minimum x-value will be the vertices of the diagonal of a polygon and the bounding box should not include this diagonal.

The next phase was to scale out the sorted coordinates to track an object. One way to do this is to simply multiply all x-coordinates with width scale factor and y-coordinates with height scale factor. However, when we did this, the scaled bounding box was not around the object. Instead, it appeared somewhere else in the image. To solve this problem, we wrote another algorithm that calculates the center of the QR code and uses it to scale back the coordinates in the right place in an image.

Another challenge we faced was decoding the QR codes in real-life scenarios. Detecting a QR code is easier than decoding it. A lot of factors can hinder the performance of a decoder, for example, camera quality, lighting, speed of movement, etc. Initially, we implemented the system using four QR codes pasted on an object, and our algorithm required all four of them to be decoded at the same time to detect and track that object in real-time. The algorithm worked perfectly well in ideal conditions. However, when the tool was moved around, almost always, at least one QR code out of the four was not decoded. This caused the object tracking to fail. Also, having to calibrate four different QR codes for a single face of an object would have been a nightmare.

To overcome this problem, we changed and optimized our algorithms to use only one QR code instead of four. To overcome the problem of failed decodes, our algorithm uses the concept of the region of interest. It means that we assume the object will be in our region of interest for almost a second, and we keep tracking it in the same area. To make this possible we save the area inside a bounding box and call it the region of interest which is overwritten after every successful decode. When the system is able to detect the QR code but is not able to decode it, we use the last saved region of interest and continue tracking the object. This has greatly improved our tracking and is especially effective in real time scenarios. Fig 1 shows the overall working of our system.

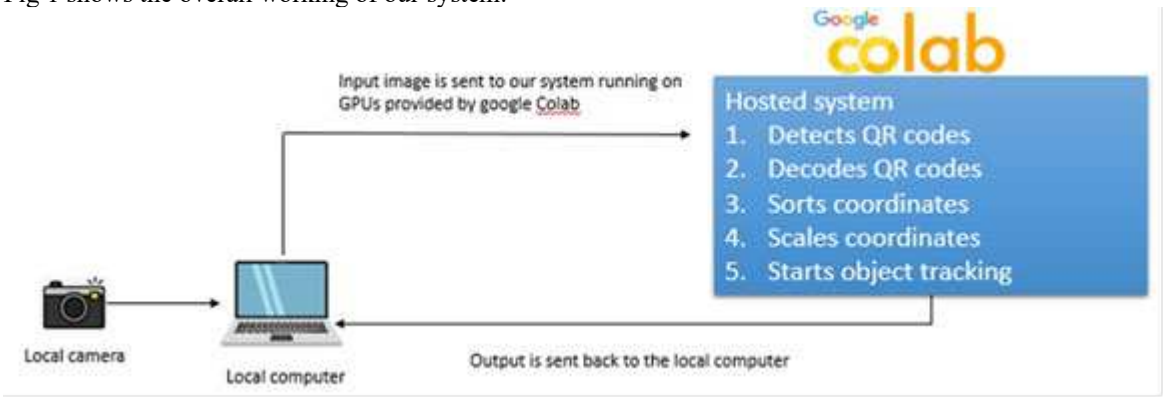


Fig 1: Overall system setup and working

## 3.2Dataset

To train this model, we required annotated set of images of QR codes that can be fed into the model to train it to detect QR codes. The easiest way would have been to generate hundreds of QR codes using a simple python script. However, we needed images of QR codes in different settings, images of distorted QR codes, and images with multiple QR codes. This means we needed a diversified dataset to make detections robust for real-time and real-world settings. To accomplish this, we used a publicly available dataset [6] along with our own custom dataset. [6] contains around 760 annotated images. However, the images are annotated for both QR codes and FIP patterns. Our use case only required QR codes and not FIPs. Also, these annotations were not in the YOLO format of text files. They are available in a different format with $x_{max}$, $y_{max}$ $x_{min}$, and $y_{min}$ coordinate values of the position of an object in the image, Fig 2. These annotations are stored in *.csv files. To be able to use this dataset, we converted the annotation files in a format accepted by YOLO. We converted the coordinate values available into (x-center, y-center), which are the coordinates of the center of the object and the object's height and width. Then we normalized these values. Normalization means that x-coordinate and width is divided by the width of the entire image and the height and y-coordinate is divided by the height of the entire image. This is repeated for every QR code in every image. All these values are saved in a text file, and one text file is saved for every image.
However, images in this dataset [6] are not as diversified for our use case. There are only a few images with multiple QR codes and a few images containing distorted or bent QR codes. To enrich the overall data, we also created our own custom

dataset. We created our custom dataset to initially train our model and then used the publicly available dataset to further train the model on previous training weights. For comparison, we also trained our model only using [6].
Our custom dataset includes approximately 500 annotated images of QR codes from various angles and in various environments. We have annotated these images with labelImg which saves the x and y coordinates of the center, height, and width of the object in the image, all in a text file in a normalized form. This has been explained above.
The process of acquiring and preparing our custom dataset is described as follows: The data has been collected from google images using the chrome extension called Image downloader by Pact Interactive. It is an interactive tool that comes with a size funnel which is particularly useful for filtering images of compatible sizes.

Darknet can process either .jpg or .jpeg image formats. Downloaded images are of different formats like .png,. jffif, etc. To convert the images in .jpg format, we used a python script. Once all the images have been converted to .jpg format, we can now label them by using labelImg [4]. It is an open-source tool written in python and uses QT for its UI. The data set has been split in 80-20 ratio where 80% of the data is set aside for training and 20% for validation.

### 3.3Training

We used transfer learning for training YOLOv4. It means that we reused the pretrained YOLOv4 on MS COCO dataset. YOLOv4 is configured to detect 80 different classes of objects. However, for our use case, we need to train it on only 1 class i.e., QR codes. For this reason, we reduced the number of filters in convolutional layers to reduce the complexity and number of parameters for our training. The formula to calculate the number of filters in accordance with the classes of objects is mentioned below [16][17]. This approach reduces the resources and amount of labeled data required to train a new model. We did not have to start training from scratch, which would have increased the consumption of valuable GPU resources.

### 3.4Terminology

1. $x_{max}$, $y_{max}$ $x_{min}$, and $y_{min}$: Coordinates show the min and max values on x-axis and y-axis, respectively, in Fig 2.
2. Region of interest: When a QR code is successfully decoded and a bounding box is scaled and drawn around the corresponding object, its coordinates are saved, and the area between them is called the region of interest. It is saved after every successful decoding but used only if decoding fails but detections are successful.

3. Center of QR code: This is an approximate value of the center of a QR code. The center's coordinates are calculated as:

$$\text{Xcenter}= \frac{x_1+x_2+x_3+x_4}{4} \text{,Ycenter} = \frac{y_1+y_2+y_3+y_4}{4}$$

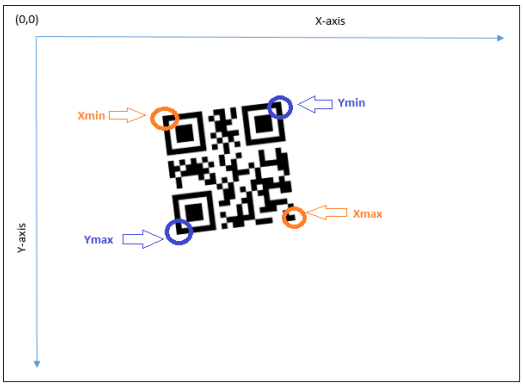


Fig 2: Xmax (greatest value of x-coordinate out of 4 coordinates of the QR code), Ymax (greatest value of y-coordinate out of 4 coordinates of the QR code) Xmin (minimum value of x-coordinate, and Ymin (minimum value of y-coordinate out of 4 coordinates of the QR code).

### 3.5. Implementation

This system draws bounding boxes around the detected object and tracks it while it is in motion. The bounding box also tilts when the object is tilted.

1. Generate a QR code and encode the following information:
    a. What is the object? Example beaker, pipette, etc.
    b. Scale factor for:
        i. Width of the object: Ratio of the width of the object: width of the QR code

        $$\text{X-axis Scale factor} = \frac{w_0}{w_Q}$$

        ii. Height of the object. Ratio of the height of the object: height of the QR code

        $$\text{Y-axis scale factor} = \frac{H_0}{H_Q}$$

2. *Detection:* After inference, the QR code is detected, and its (X-center and Y-center coordinates, height, and width are returned.
3. *Decoding:*
    *a.* The detected part of the image is sent to the PyzBar for decoding. After decoding, it returns the encoded information along with four coordinates of the QR code.
    *b.* We did not send the entire image to PyzBar to reduce decoding time which optimizes the performance in real-time.
    *c.* PyzBar returns the decoded information mentioned in step 1 of this algorithm. It also returns the four coordinates of the QR code's vertex.
    *d.* However, these coordinates are not in reference to the whole image, only in reference to the QR code's image that was sent to PyzBar.
    *e.* To rectify this problem, we have subtracted the original $x_{min}$ and $y_{min}$ from x and y part of all four coordinates, as shown in Fig 3.
    f. Also, the system creates a log file for detection and decoding with timestamps.

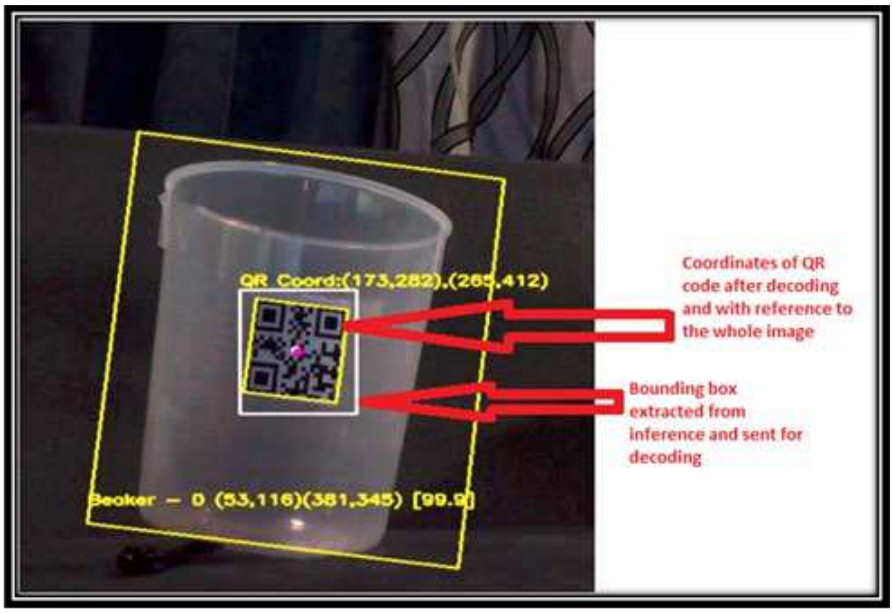


Fig 3: Two sets of coordinates, one after inference and one after decoding.

4. *Sort coordinates:*
    a. The next step is to determine how to draw lines correctly between the four given coordinates. We need to make sure not to draw lines along the diagonals. This can only be done by understanding which coordinates are which.
    b. To achieve this, we have sorted all four coordinates based on their x coordinate's value in ascending order.
    c. We now have sorted coordinates of the 4 vertices of the QR code as shown in Fig 4.

$$V_0 < V_1 < V_2 < V_3$$

5. *Draw bounding box around QR code:*
    a. The idea is that coordinates with $x_{max}$ and $x_{min}$ will never be joined with a line, and similarly, $y_{max}$ and $y_{min}$ will never be joined as these coordinates will always be the diagonals of the bounding box.
    b. Lines are drawn according to the sorted list of the coordinates. Lines are drawn between ($V_0$, $V_1$), ($V_0$, $V_2$), ($V_2$, $V_3$) and ($V_3$, $V_1$). Fig 4 illustrates how to draw a bounding box.
    c. Also in Fig 3, yellow bounding box around QR code is a real-time example.
    d. Fig 5 clearly shows how bounding boxes of this algorithm show the tilted object whereas Yolov4's box does not; it is always a perfectly straight square or rectangle.

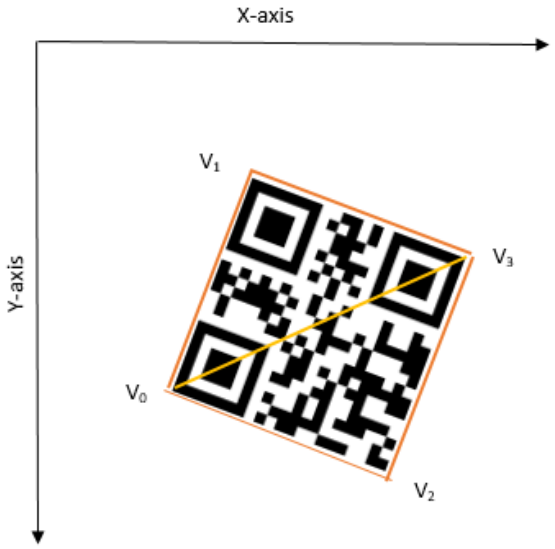


Fig 4: sorted vertex positions and illustration of which Coordinates should be joined by lines to create a bounding box. Yellow line shows that we shouldn't join V0 and V3 or V1 and V2

6. *Scaling:*
    a. So far, we have achieved to draw a correct bounding box around the QR code. However, our final goal is to do this for the entire object. This bounding box is drawn around the QR code pasted on the object but not the object itself.
    b. Therefore, the next step is to scale these coordinates to correctly position the object. To scale this bounding box, we first calculate the distance between

diff_x = (Xcenter* X-axis Scale factor) – Xcenter
diff_y = (Ycenter* Y-axis Scale factor) – Ycenter

These two values will be used to get the scaled box in a position that is in alignment with the QRcode's center.

c. For each vertex:
    i. Multiply X coordinate with X-axis Scale factor and subtract the diff_x
    scaled_x-coordinate= (x-coordinate * X-axis scale factor) - diff_x
    ii. Multiply Y coordinate with Y-axis Scale factor and subtract the diff_y
    scaled_x-coordinate= (x-coordinate * X-axis scale factor) - diff_y
d. We have the scaled coordinates and can draw a line between them as described in step 5 and Fig 5.

7. *Decoding Problem:*
    a. Steps 3 to 6 will work perfectly well as long as the QR code is decoded in every frame. However, successfully decoding a QR code depends on a lot of factors like correct camera focus, distance from the camera, amount of light in the room, reflecting lights, etc. In real-time and real-life scenarios, it is impossible to decode in every frame and could lead to miss detection of the object.
    b. To solve this problem, we have used the concept of the region of interest. After every successful decoding, the region of interest is updated with the area between the bounding boxes. As soon as there is a detection but no decoding, the region of interest from the previous frame is used to approximate the position of the object, and there is no break in object tracking.
    c. However, this is done for a maximum of 1 second because after a certain time, the object may change its position and the previously stored region of interest becomes obsolete.

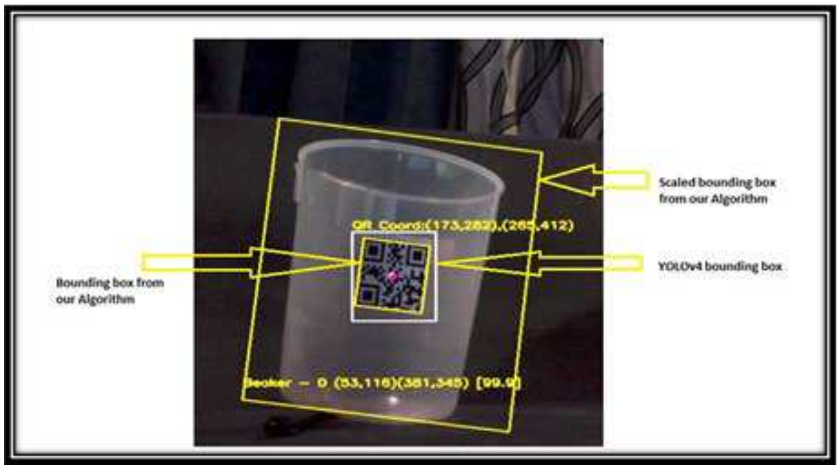

Fig 5: Different bounding boxes

## 4. RESULTS ANALYSIS

### 4.1. Model Evaluation

#### 4.1.1. Performance Evaluation of YOLOv4 and other SSD algorithms

To compare and evaluate the performance of YOLOv4 with other SSD algorithms, we trained our model using the [6] dataset. The dataset [6] is divided into 80% training data, 10%testing data, and 10% validation data. We preserved this division and trained YOLOv4 using the exact same division of their dataset. Following this, we compared the results presented in paper [6] with the same dataset. Fig 6 represents these results. YOLOv4 has performed approximately 30% better than MobileNet, and 40% better than VGG16 and RestNet50.

| Object Detection models | mAP@50% threshold |
|---|---|
| **YOLOv4** | 94.17% |
| **MobileNet** | 72.7% |
| **VGG16** | 67.70% |
| **RestNet50** | 67.1% |

Fig 6: Comparing the performance of different 1-stage algorithms trained on [6] dataset. Results show that YOLOv4 has the highest performance

### 4.1.2 Evaluation of QR decoding performance in terms of frames per second

We compare some existing decoders in Fig 7, available in the open-source community, with our proposed algorithm that includes inference of QR code, decoding of QR codes, processes the decoded information to detect and track an entirely different object, and also takes care of missed decoding by using object's region of interest, all in real-time.
At the time of recording, these are the numbers recorded and are shown in the following images. All measurements for comparison are performed on Tesla T4 GPU.

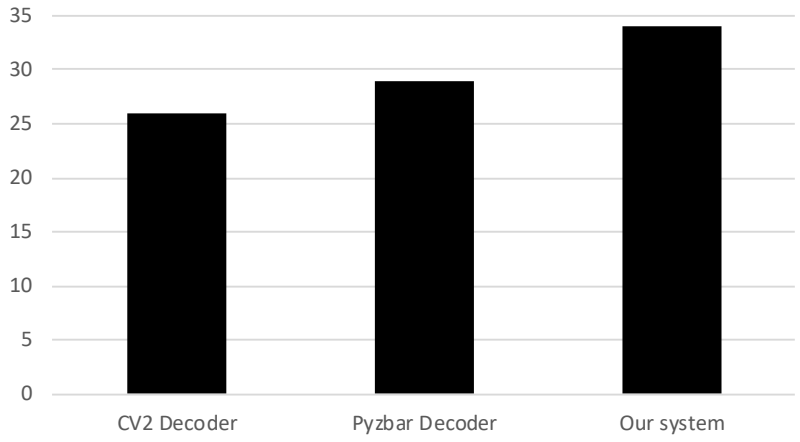


Fig 7:Peformance Comparison (Frame Per Second)

Our algorithm uses PyzBar for decoding but with optimizations. It first performs inference and then sends only that part of the image to the decoder, whereas PyzBar decodes the entire image, which makes it 17% slower.
In conclusion, this is not a direct comparison as our algorithm requires much more computation because of all the additional features mentioned above, and yet it is 30% faster than CV2 open-source decoder[5] and 17% faster than the off-the-shelf PyzBar decoder[2].
**Note: All the inference and experiments were performed on Google Colab, which is a Jupyter notebook service hosted by Google. The camera used was connected to our local system, the images were sent over the network to Colab where all computations were carried out, then the results were sent back over the network. This caused latency, and the processing was reduced to an average of 2 FPS. However, we calculated the frames per second being processed on Tesla T4 GPU provided by Colab, in short, we removed network latency in our calculations of the performance of our system, CV decoder, and off-the-shelf PyzBar.

#### 4.2. Dataset Evaluation: YOLO training results on different datasets

Following are the three types of datasets that we used for training the YOLO model:

1. **Custom dataset:** We first trained YOLOv4 with the custom dataset that we created, as presented in Fig 6.
2. **Publicly available dataset[6]**: We also trained our YOLOv4 model on this dataset, and the results are shown in Fig 8.
3. **Enhanced dataset (Custom Dataset + Publicly available dataset)**: To increase the overall precision of our model first trained YOLO with our custom dataset and used those weights to further train it with the publicly available dataset as shown in Fig 8.

| Metrics | Our custom dataset | [6] dataset | Enhanced Dataset |
|---|---|---|---|
| **mAP @50% threshold** | 91.47% | 94.17% | 93.82% |
| **Average IOU** | 74.9% | 81.38% | 77.45% |

Fig 8: Metrics after training YOLOv4 with different datasets

### 4.3. Lab tools detection and tracking results

The implemented method was tested with multiple lab tools for an educational science lab, and the results in Fig 9 and Fig 10 show that the lab tool is detected correctly and the QR markers are tracked accurately as shown by tilting a beaker in Fig 10. We have also tested the model with multiple lab tools in various conditions, including lab tools being side by side, being in the same scene with distance, as well as overlapping bounding boxes where the lab tools are still detected and tracked accurately.


Fig 9: Detection from our custom model. Top left value represents the frames being processed per second.

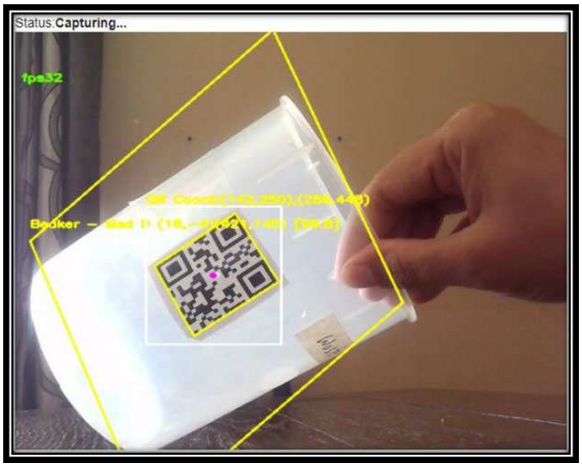


Fig 10: Tilted beaker being detected

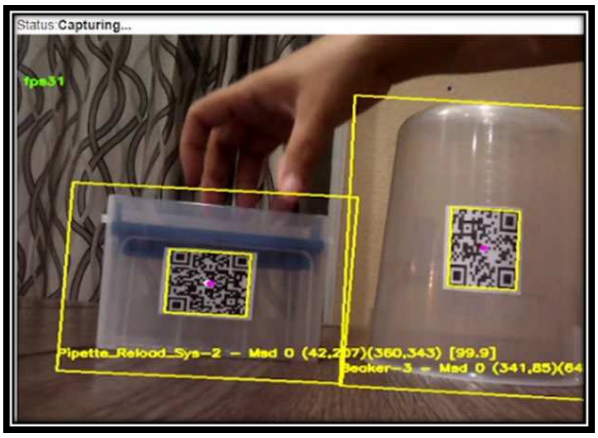

Fig 11: Multiple lab tools detected side by side.

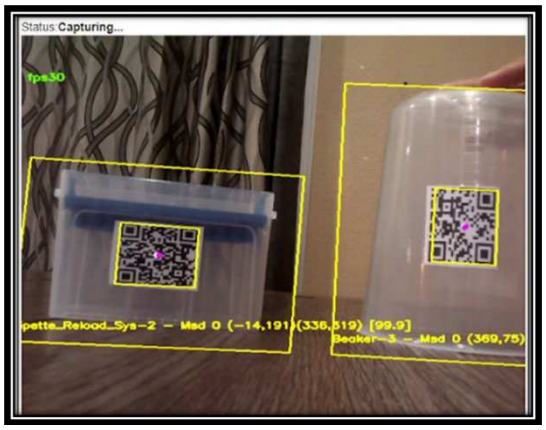

Fig 12: Multiple Lab tools detected with distance between them.

Fig 13: Multiple tools being detected with one in the background and overlapping bounding boxes.

## 5. CONCLUSION

This project demonstrates a unique way to detect and track objects in real time without having to go through the process of creating datasets and training models to do so. This is made possible by encoding a QR code with calibrated information about the object it is supposed to detect and track. Our algorithm decodes the QR code, extracts its coordinates, and then extrapolates all the available data to create a bounding box around the object for tracking. We have also implemented a method where in the case of failed decoding, the system can still approximate the position of the object by referring to its previously saved region of interest. We have also demonstrated how an object's titled orientation can be captured by a bounding box which is an improvement to the existing outputs of YOLOv4; Fig 3 demonstrates this feature. This means that whenever we have to detect and track a new object, all we have to do is generate a calibrated QR code and paste it on that object. Our algorithm will start tracking it in real time.

YOLO4 model has performed approximately 30% better than MobileNet, 40% better than VGG16 and RestNet50 when compared using the same publicly available dataset[6]. The throughput of our project in terms of frames per second is 34 fps which is 31% faster than Open CV decoder and 17% faster than PyzBar. When comparing datasets, our custom dataset yielded a 91.7% mAP on YOLOv4, dataset [6] yielded 94%, and first training the model on our custom dataset and using those weights to further train that model with the other dataset yielded 93.8% mAP. We have improved decoding performance by extracting QR code from the entire image and sending only a fraction of the whole image containing the QR code to PyzBar decoder. This technique has helped us improve the overall performance of our system.

## REFERENCES


[1] L. Karrach, E. Pivarčiová, and P. Bozek, "Recognition of Perspective Distorted QR Codes with a Partially Damaged Finder Pattern in Real Scene Images," *Appl. Sci.*, vol. 10, no. 21, p. 7814, Nov. 2020

[2] L. Belussi and N. Hirata, "Fast QR Code Detection in Arbitrarily Acquired Images," in *2011 24th SIBGRAPI Conference on Graphics, Patterns and Images*, Alagoas, MaceiA, Brazil, Aug. 2011, pp. 281–288.

[3] MIT license, "PyzBar." [Online]. Available: https://pypi.org/project/pyzbar/

[4] "labelImg." https://github.com/tzutalin/labelImg

[5] "CV2 QR Code decoder."https://docs.opencv.org/3.4/de/dc3/classcv_1_1QRCodeDetector.html

[6] L. Blanger and N. S. T. Hirata, "An Evaluation of Deep Learning Techniques for Qr Code Detection," in *2019 IEEE International Conference on Image Processing (ICIP)*, Taipei, Taiwan, Sep. 2019, pp. 1625–1629.

[7] https://github.com/AlexeyAB/darknet#how-to-train-to-detect-your-custom-objects

[ 8] S. Srivastava, A. V. Divekar, C. Anilkumar, I. Naik, V. Kulkarni, and V. Pattabiraman, "Comparative analysis of deep learning image detection algorithms," *J. Big Data*, vol. 8, no. 1, p. 66, Dec. 2021

[9] P.-C. Wu *et al.*, "DodecaPen: Accurate 6DoF Tracking of a Passive Stylus," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, Québec City QC Canada, Oct. 2017, pp. 365–374.

[10] H. Sarmadi, R. Munoz-Salinas, M. A. Berbis, and R. Medina-Carnicer, "Simultaneous Multi-View Camera Pose Estimation and Object Tracking With Squared Planar Markers," *IEEE Access*, vol. 7, pp. 22927–22940, 2019

[11] N. Xiang, X. Yang, and J. J. Zhang, "TsFPS: An Accurate and Flexible 6DoF Tracking System with Fiducial Platonic Solids," in *Proceedings of the 29th ACM International Conference on Multimedia*, Virtual Event China, Oct. 2021, pp. 4454–4462. doi: 10.1145/3474085.3475597.

[12] F. Sultana, A. Sufian, and P. Dutta, "A Review of Object Detection Models Based on Convolutional Neural Network," in *Intelligent Computing: Image Processing Based Applications*, vol. 1157, J. K. Mandal and S. Banerjee, Eds. Singapore: Springer Singapore, 2020, pp. 1–16.

[13] R. Girshick, "Fast R-CNN," *ArXiv150408083 Cs*, Sep. 2015, Accessed: Apr. 10, 2022.

[14] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature Pyramid Networks for Object Detection," *ArXiv161203144 Cs*, Apr. 2017, Accessed: Apr. 10, 2022. [Online].

[15] K. He, G. Gkioxari, P. Dollar, and R. Girshick, "Mask R-CNN," in *2017 IEEE International Conference on Computer Vision (ICCV)*, Venice, Oct. 2017, pp. 2980–2988.

[16] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," *ArXiv150602640 Cs*, May 2016, Accessed: Apr. 12, 2022. [Online].

[17] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia, “Path Aggregation Network for Instance Segmentation,” *ArXiv180301534 Cs*, Sep. 2018, Accessed: Apr. 14, 2022. [Online].
[18] K. Suresh, M. Palangappa, and S. Bhuvan, “Face Mask Detection by using Optimistic Convolutional Neural Network,” in *2021 6th International Conference on Inventive Computation Technologies (ICICT)*, Coimbatore, India, Jan. 2021, pp. 1084–1089.
[19] A. Neubeck and L. Van Gool, “Efficient Non-Maximum Suppression,” in *18th International Conference on Pattern Recognition (ICPR’06)*, Hong Kong, China, 2006, pp. 850–855.
[20] S. Tiwari, “An Introduction to QR Code Technology,” in *2016 International Conference on Information Technology (ICIT)*, Bhubaneswar, India, Dec. 2016, pp. 39–44.
[21] Sergio Garrido-Jurado, Rafael Muñoz-Salinas, Francisco José Madrid-Cuevas, and Manuel
Jesús Marín-Jiménez. 2014. Automatic Generation and Detection of Highly Reliable Fiducial Markers Under Occlusion. Pattern Recognition 47, 6 (2014), 2280–2292.