Group Number 8: Shopping Cart App for Self-checkout

Pratheek Tirunagari
Ashruj Gautam
GitHub Repository for Project: Link

Abstract

In our project, we aim to create a shopping cart mobile particularly Android application which is capable of detecting apples so that customers can use it to self checkout. For this we tried three different object detection models names: SSD (Single Shot Detector), Faster R-CNN, and YOLO (You Only Look Once). We trained these on the very famous COCO dataset made by Microsoft, given we only used the apple (class 47) of the coco dataset to train our models. Our initial results very cleared showed that YOLO outperformed the remaining two models so we decided to move forward with it. But our implementation lacked novelty. So, to address this, we tried to solve the problem of occlusion in apple detection.

For that we first designed our own custom YOLO architecture and we trained it from scratch on the COCO dataset, but the results we got were inconsistent mainly because of time and resource constrains as we lacked the hardware and time to achieve desired results. Next we wanted to check if the best of the best pretrained weights which are the industry standard have achieved any success or breakthrough in solving the occlusion problem in apple detection. For this we tried the YOLOv11x.pt which was released February 2025 and was the largest most resource hungry but the best out there but to our shock even this pretrained weight gave inconsistent results. Even though it was better than our custom architecture and self trained model it was still struggling with occlusions. We then implemented pre- and post-processing techniques like multi-scale detection, soft-NMS, etc, to enhance occlusion handling which yielded in better results, but it was still inconsistent.

Finally we achieved novelty and fulfilled the novelty criterion by solving a new challenge we identified which the inability of YOLO to detect small apples (far away from the camera). To solve this we used the MinneApple dataset, we analysed the best resolution which is efficient for detecting apples at diffrent distances like medium and far. Our finding provide insights for solving a real world problem which is the balance striking resolution between quality of detection and resources used.

1 Dataset

1.1 COCO dataset.

Description: The COCO dataset which means Common Objects in context is a dataset that has more than 200 thousand labeled images with 80 object categories. We focused only on Apples which is the class 47 according to its data.yaml file for our training and evaluation of models.

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

Data Engineering:

- **Preprocessing:** First off all the images were resized to 1280 x 1280 and then all of them were normalised this is standard practice.
- **Augmentation:** We applied augmentation techniques on our dataset like the mosaic augmentation, random flipping and then the colour space adjustments (HSV shifts) to better our datasets generalizing meaning making our dataset less similar/ predictable hence training on this will yield better generalised models.
- **Class Filtering:** As mentioned before COCO dataset has many around 80 different classes and we wanted only class 47 which is the apple class. So we filtered the dataset so that we only get apple classes and can begin training our models.

1.2 MinneApple Dataset

Description: This is a specialised dataset which contains only images of apples but the apples are in the orchards, it contains apples many apples which are particularly small in the images meaning they are far away from the camera which helped us in better judging and solving our resolution issue.

Data Engineering:

- **Resolution Analysis:** Using this dataset we first tested detection and performance of our model on 7 different resolutions so we had to change every image in the dataset into 7 different resolutions from 320 x 320 to 1280 x 1280, to detemine which input size is the best meaning optimal for distant apples.
- **Distance Estimation:**Since we also wanted to categorize our apples into distance of medium or far based on the bounding box size and the image dimensions. Because we wanted to give recommendation of input size according to distance and not just one size fit for everyone.

2 Model Description

2.1 Custom YOLO Architecture

Our custom YOLO architecture is designed specifically for apple detection with modifications to improve the overall performance mainly on the occluded and small apples. Give below is the architecture in detail:

Backbone: The backbone is mainly responsible for extracting the features the modifications we made to the backbone are:

- **Larger Receptive Field:** We increased the SPPF kernel size from 5x5 to 7x7 so that it can better capture the contextual information as this is important to detect the occluded apples.
- **Enhanced Feature Extraction:**We added the C2PSA which means Cross staged partial spatial attention blocks we made it to 3 repeats instead of the 2 because this will improve our feature fusion.
- **Multi-Scale Processing:** Before in the standard yolo architecture, P3/8 was used we changed it to P5/32 feature pyramid as this is better for detecting apples at different scales.

Neck: The mainly combines the features that are extracted from diffrent layers to improve the detction:

- In the neck we first modified the upsampling and we used the nn.Upsample with nearest neighbor interpolation as this is better and produces smoother feature maps
- Next we used concatenation based fusion meaning this combines the features taken from different depths like P3, P4, P5 to retain the details for smaller apples so that they are detected.

Head: The head mainly predicts the bounding boxes and also predicts the class probabilities:

- First off we implemented the **Multi-Scale Detection** meaning it detects apples at three different scales:
  - P3/8 - For small apples this has higher resolution and is best for apples which are very small in the image meaning they are far away from the camera
  - P4/16 - These are for medium sized apples are have a fairly balanced detection
  - P5/32 - These are for large apples meaning they are close up photos or apples.

Key Changes we made to the architecture:

1. We increased the kernel size to SPPF 7x7 so that it helps detect partially hidden apples which will help in occlusion.
2. Addition of C2PSA blocks which improve the feature fusion and can help in detection the overlapping apples
3. Finally we optimised the architecture for a single class as we only need to detect apples we removed a lot of complexity which is unnecessary for our use case so as to reduce the training time and computational resources taken.

2.2 Models Tried and why YOLO is the Best

We tried and evaluated three models which are:

Model	Strengths	Weaknesses	Why YOLO Won
SSD (Single Shot Detector)	Fast inference, simple architecture.	Struggles with small/occluded objects, low mAP.	Too inaccurate for self-checkout.
Faster R-CNN	High accuracy, good for occlusion.	Very slow (not real-time), complex two-stage pipeline.	Unsuitable for real-time applications.
YOLO (Our Custom architecture)	Real-time speed, good balance of accuracy & speed, handles occlusion better with our modifications.	Requires tuning for small apples (solved via resolution analysis).	Best for self-checkout due to speed + accuracy.

Table 1: Comparison of Object Detection Models for Self-Checkout System

2.3 Understanding the YOLO Algorithm

YOLO (You Only look once) is a object detection model which is single stage this processes the images in one forward pass and how does it work ? The answer to this is given below in detail:

Step 1: Grid division- Every input image is first divided into SxS grids, and then each grid cell predicts their own B which is the bounding boxes and their corresponding class probabilities.

Step 2: Bounding Box Prediction- Each and every bounding box has coordinates and their corresponding confidence score which means how probable or sure is the model that there is an apple inside this bounding box. Next it also gives the class probability which is always one in our case because we are only detecting apples but if we had multiple classes then this would be useful.

Step 3: Non Max Suppresion- This mainly filters all the duplicate detections by selecting and keeping only the box which has the highest confidence and discards the rest. Our soft-NMS improves occlusion handling by mainly reducing the confidence scored for overlapping boxes as for occlusion if we discard them we will fail in our task.

Step 4: Multi-Scale Detection- We use the feature pyramids (P3, P4, P5) to detect our apples of various sizes. For our custom architecture it enhances this with additional upsampling and also uses C2PSA blocks.

3 Loss Function

3.1 What is the loss function you choose ?

The main loss function that we used in our custom YOLO architecture is a **composite loss function** this consistes of three main components which are:

1. Bounding Box Loss (CIoU Loss): This loss measures the accuracy of the bounding boxes our model predicted using the complete Intersection over union (CLoU) and this CLOU takes into account the overlap area, the center distance and the aspect ratio.
2. Classification Loss (Focal Loss): This addresses the class imbalance by mainly focusing on the hard to classify examples thus reduces the effect that the easily classified background samples will have on the training of the model.
3. Distribution Focal Loss (DFL): This betters the localization precision by mainly bettering or optimizing the probability distribution of all the bounding box coordinates.

3.2 What other loss functions did you try?

We tried many different loss functions during the experimentation phase, some of them are:

1. MSE loss for the bounding boxes: First during our initial days we tested our model with this loss function but we quickly discarded it because of poor convergence and sensitivity to outliers.
2. Cross Entropy loss: We used this during our experimentation phase but we replaced it with our focal loss to lessen the class imbalance issues that we were phasing.
3. DIOU Loss: This is a much simpler version or variant of the CIOU this one ignores the aspect ration which made it perform worse when we tested it.

3.3 Innovation on the loss function?

We made two innovations in the loss function which are:

1. **Adaptive Loss weighting:** This means that we dynamically adjust the weights of the CIOU, Focal and the DFL losses during our training based on the magnitudes of our gradients, which I believe improves stability.
2. **Occlusion Aware Focal Loss:** We also modified our focal loss so as to penalize or punish the model more heavily if it misclassifies partially occluded apples.

4 Optimization Algorithm

4.1 What optimization algorithm did you use?

We used the AdamW which is Adam with Weight Decay as our main optimizer, we configured it with the following hyperparameters:

- **Initial Learning Rate**(lr0) which we set to 0.0001
- **Final Learning Rate**(lrf) which we set to 0.01 with the cosine decay
- **Weight decay** is set to 0.0001 this prevents overfitting by punishing or peanalizing large weights basically regularization.
- **Momentum:** This is set to 0.937 which smooths gradient updates.
- **Batch Size**: We set our batch size to 4 because of GPU constraints we had.

4.2 What other optimization Algorithms have you tried?

We tried a few other optimization algorithms which are:

1. Stochastic Gradient Descent (SGD) with momentum as 0.9: The good part about this was its simplicity but although theoretically it guarantees results it required a lot of manual rate tuning and also it had a slower convergence for more epochs.
2. Adam (without the weight decay) also like the above it is more simplet and had a faster convergence at the start but the issue came later becaues it overfits more easily and this happened because of the improper weight decay handling.
3. Lion (Lion optimizer) the good part is that is very memory efficient and was theoretically very good but the issue became that it is unstable with small batch sizes and as our batch size is 4 it lead to very erratic loss curves and we decided to not go with this.

4.3 Innovations on the Optimization Algorithm

We enhanced the optimization for occlusion handling, we implemented:

1. **Gradient Based Learning Rate Scaling:** This means that our model dynamically adjusted our hyperparameter lr0 for our bounding box prediction when ever an occlusion was detected. This was done when ever we get high IOU and that conflicts with our training data.
2. **Warmup with occlusion aware scheduling:** Next we changed our warmup epochs hyperparameter to increase the learning rates slower for samples with occluded apples this is detected via label overlaps.
3. **Gradient Clipping with adaptive thresholds:** We clipped the gradients at max norm = 1.0 for classification head and for the bounding box we set the same at 2.0 to particularly handle the noise that came as a result of occlusion.

5 Metrics and Experimental Results

5.1 Evaluation Metrics we used

The evaluation metrics that we used are as follows:

- **Precision:** This simply measures the accuracy of the positive predictions which means that how many apples we detected were actually ground truth apples.
- **Recall:** This measures the ability to detect all the ground truth apples meaning actual apples.
- **mAP50:** This is the mean average precision at IOU (intersection over union) with a threshold of 0.5, this evaluates detection accuracy across all confidence thresholds.
- **mAP50-95:** This is the mean average precision at IOU (intersection over union) with a range of threshold from 0.5 to 0.95, this evaluates detection accuracy across all confidence thresholds.
- **Box Loss:** This is the regression loss for all the bounding box coordinates.
- **Cls Loss:** This is the classification loss for object confidence.
- **DFL Loss:** This is the distribution of the focal loss for the bounding box refinement.

5.2 SSD (Single Shot Detector) Results

We trained our SSD on the following hyperparameters: Batch size: 32, Learning rate: 0.0001, Weight decay: 1e-4, Optimizer: AdamW (with eps=1e-8), Number of epochs: 50, Early stopping: 12 epochs, Gradient clipping: 0.5.

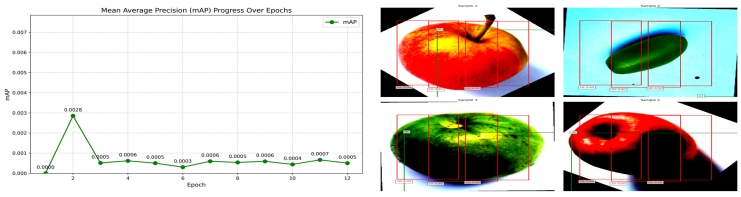

Figure 1: (Left) Visualization of the training of SSD. (Right) Experimental result for SSD.

5.3 Faster R-CNN Results

Validation results: Initial mAP: 0̃.005 (0.5%), Peak mAP: 0̃.030 (3.0%) at epochs 2-3, Secondary peak: 0̃.030 (3.0%) at epoch 7, Final mAP: 0̃.015 (1.5%). Overall trend: Initial improvement, fluctuation, then decline.

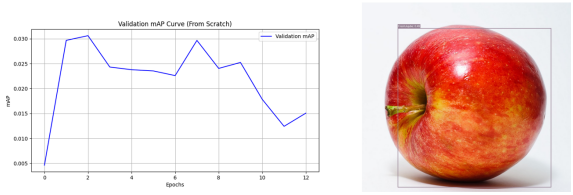

Figure 2: (Left) Visualization of the training of Faster R-CNN. (Right) Experimental result for Faster R-CNN.

5.4 Custom YOLO Architecture Results

We trained our custom architecture of YOLO from scratch without using any pretrained weights on the COCO dataset (specifically class 47 for apples). Gist of results (see `results.csv` for details):

Epoch	Precision	Recall	mAP50	mAP50-95	Box Loss	Cls Loss	DFL Loss
1	0.015	0.043	0.007	0.002	3.434	3.881	3.374
50	0.371	0.345	0.279	0.180	1.191	1.656	1.522
100	0.347	0.298	0.237	0.154	0.916	1.042	1.273
131	0.392	0.298	0.276	0.162	0.819	0.814	1.191

Table 2: Model performance metrics over training epochs.

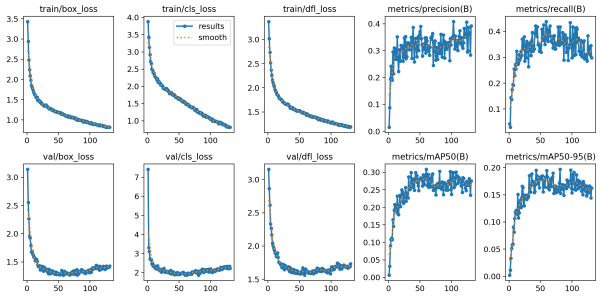

Figure 3: Training visualization of our custom YOLO architecture.

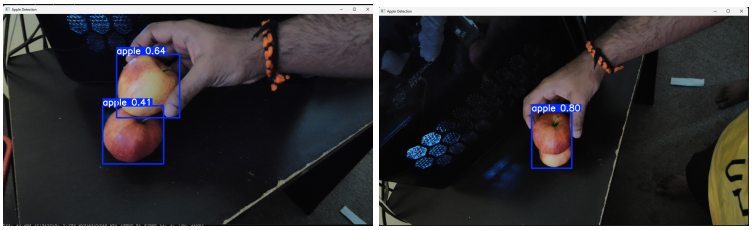

Figure 4: Experimental results for Custom YOLO Architecture.

5.5 Pretrained YOLOv11x with Post-Processing

Post-processing techniques:

- Multi-Scale Detection: inference scales `[0.5, 0.75, 1.0, 1.25, 1.5]`
- Soft NMS: Gaussian penalty for overlapping boxes.
- Sliding Window: for images >1500 pixels, 50% overlap, windowed detection.

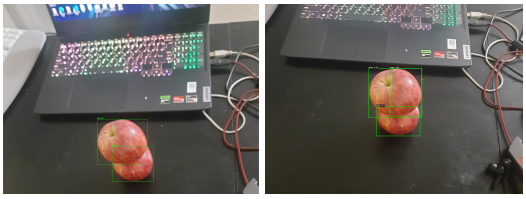

Figure 5: Post-processing experimental results.

5.6 Resolution Analysis (MinneApple Dataset)

To detect small, distant apples, we tested different resolutions (320x320 to 1280x1280). Recommended minimum resolutions:

- Far distance: 416x416
- Medium distance: 320x320

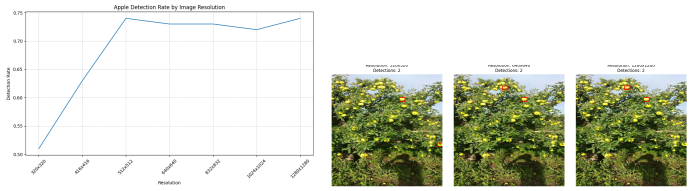

Figure 6: Resolution analysis visualization and results.

Contributions of Group Members

Both of us woked equally (50% each) on the project by deviding our tasks and working on them simulaneously to save time and resources here is the breakdown of our work:

Task	Pratheek	Ashruj
Research & Planning	YOLO architecture, occlusion solutions	SSD/Faster R-CNN benchmarking
Custom YOLO11x	Designed backbone (yolo11.yaml)	Hyperparameter tuning (args.yaml)
Training & Evaluation	Trained models, analyzed results.csv	Tested pretrained YOLOv11x
Occlusion Handling	Multi-scale detection (kong.py)	Soft-NMS & clustering
Resolution Analysis	Core testing code (apple_resolution_analysis.py)	Visualization & recommendations
Report Writing	Model/Loss sections	Abstract/Optimization sections

Table 3: Contribution Table for the Project

References

[1] COCO Dataset | Link
Minneapple Dataset | Link
Lecture Notes From Piazza