Dirt Detection for Cleaning Robots

Yutong Chen, Jiaming Yu and Hailin Chen

***Abstract*— In this report, we present a YOLOv8-based visual recognition model designed specifically for cleaning robots. The primary goal of our model is to effectively distinguish dirt from common office items.**

Our model is built upon the YOLOv8 architecture, which is renowned for its impressive speed and accuracy in real-time object detection tasks. By using YOLOv8 as a foundation, we trained our model using a comprehensive dataset comprised of annotated images of dirt and various office items. The system's effectiveness has been extensively evaluated through various performance metrics, such as precision, recall, and F1 score.

1. MOTIVATION

The functionality of cleaning robots is often hindered by the limited capability of distinguishing between different types of objects. This lack of precise recognition can lead to erroneous interactions with non-dirt objects, thereby decreasing cleaning efficiency and potentially causing damage.

Our motivation stems from this need. We strive to enhance the autonomous capabilities of cleaning robots, allowing them to identify and differentiate between dirt and other objects more accurately. Using the state-of-the-art real-time object detection model, we aim to train our cleaning robots with a robust ability to recognize various types of dirt, separate from common office items.

2. MODEL

We firstly need to choose a proper model for the detection task. The chosen model should have the following characteristics:

- Fast.(For real time detection)
- Precise.(For small objects)
- Low computational demands

Hence, we choose one of the most popular real-time detection model, YOLO, for our task.

2.1 YOLO MODEL

YOLO stands for "You Only Look Once." It is a popular object detection algorithm in the field of computer vision. As opposed to traditional object detection models that involve two-step processes, YOLO applies a single, unified model to this task, which makes it extremely fast and efficient. Instead of using a system that first identifies regions of interest and then classifies those regions, YOLO performs both tasks in a single pass. This dramatically increases speed. Figure 1 shows the underlying structure of YOLOv1. The YOLOv1 detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the features space from preceding layers.

Instead of using a system that first identifies regions of interest and then classifies those regions, YOLO performs both tasks in a single pass. This dramatically increases speed.

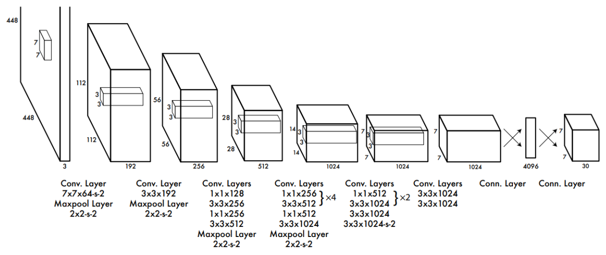


Fig. 1. Structure of YOLOv1 [1]

2.2 YOLOv5 vs YOLOv8

YOLOv5 and YOLOv8 are the two most stable and sophisticated models of all the YOLO models and outperform any other versions. Before any further implementation we compared YOLOv5 and YOLOv8 to see which model fits our requirements the best.

Compared to previous versions, YOLOv8 introduces numerous improvements such as a new neural network architecture that utilizes both Feature Pyramid Network (FPN) and Path Aggregation Network (PAN) [2], [3]. The PAN architecture aggregates features from different levels of the network through skip connections. By doing so, the network can better capture features at multiple scales and resolutions, which is crucial for accurately detecting objects of different sizes and shapes. The structure of PAN is shown in Figure 2. Since our major class to be detected is dirt, which is usually in a smaller scale than other common objects, YOLOv8 is perfectly suitable for our task.

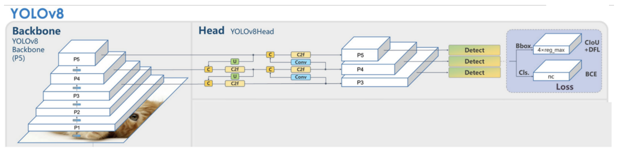


Fig. 2. Structure of PAN [2]

We run the training with the same dataset (1156 training pictures and 80 testing pictures) and the same epochs (30 epochs). The results are shown in Figure 3 and Figure 4. We can see that the mAP of YOLOv8 is 0.911, which is 0.002 higher than YOLOv5. Therefore, we choose to use YOLOv8 for further applications.

3. IMPLEMENTATION

3.1 Dataset Transformation

Compared with YOLOv5, YOLOv8 has a lot of upgrades. Firstly, we need to convert the label into the form of YOLO.

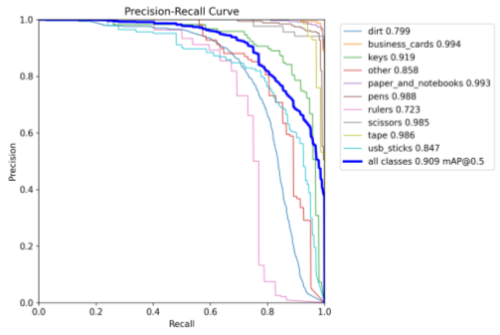


Fig. 3. 1156 training set with yolov5

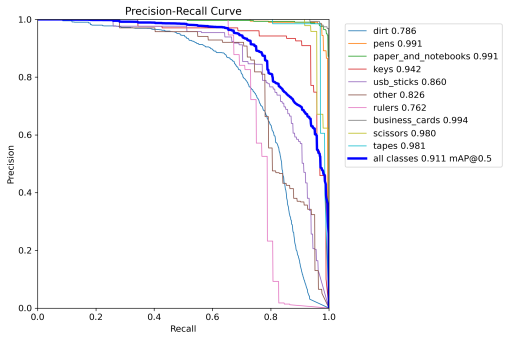


Fig. 4. 1156 training set with yolov8

The original data label format is as shown in Figure 5. All the entries of the dataset samples are stored in a single txt file. The coordinates of the bounding box is the pixel locations of the left upper and right under corner of the box. And the label format of YOLO is shown in Figure 6, in which X and Y are the center of the bounding box relative to the width and height of the image, Width and Height are the width and height of the bounding box (again, relative to the width and height of the image). So we use two simple Python programs (`prepare_data`) to change the original label format into the YOLO label format.

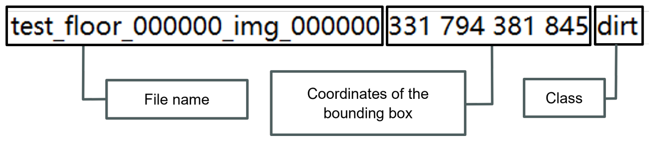


Fig. 5. The original format of labels

3.2 *YOLOv8 Implementation*

The whole package of YOLOv8 has fewer folders, and it is convenient for training and deployment, which is shown in Figure 7. After we understand the structure of YOLOv8, we can start to implement it with our own dataset.

We need to define the dataset that we transformed to the YOLOv8 format before in the folder `dataset`. In addition, the address of the dataset and the names of classes also need to be redefined in the file `mydata.yaml`, which is shown

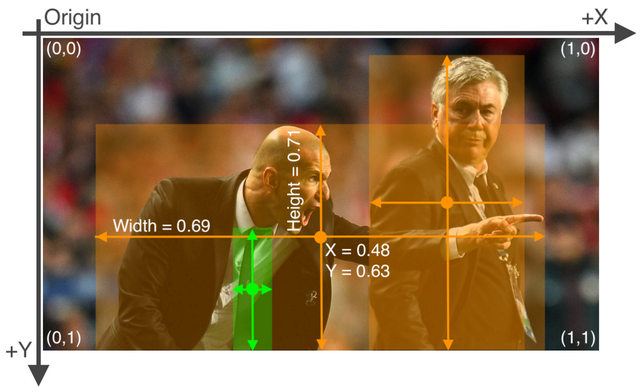


Fig. 6. The format of YOLO labels [1]

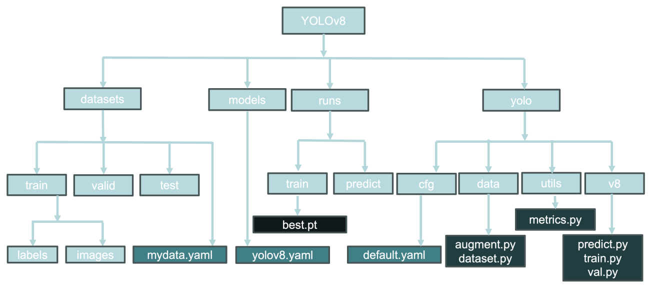
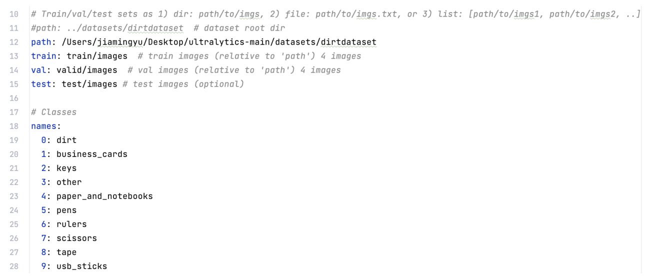


Fig. 7. YOLOv8 Structure

in Figure 8, so that the model can smoothly read the dataset and classify it correctly during training.



```
# Train/val/test sets as 1) dir: path/to/imgs, 2) file: path/to/imgs.txt, or 3) list: [path/to/imgs1, path/to/imgs2, ..]
#path: ../datasets/dirtdataset  # dataset root dir
path: /Users/jiamingyu/Desktop/ultralytics-main/datasets/dirtdataset
train: train/images  # train images (relative to 'path') 4 images
val: valid/images  # val images (relative to 'path') 4 images
test: test/images # test images (optional)

# Classes
names:
  0: dirt
  1: business_cards
  2: keys
  3: other
  4: paper_and_notebooks
  5: pens
  6: rulers
  7: scissors
  8: tape
  9: usb_sticks
```

Fig. 8. mydata.yaml

Then we can input the command in the terminal to train the model with our own dataset. The model type, the number of epochs, and the learning rate can also be modified in the command:

```
yolo train data=/datasets/mydata.yaml
model=yolov8n.yaml pretrained=yolov8n.pt
epochs=3 batch=8 lr0=0.01
```

After training, the results and the weights file `best.pt` are stored in the folder `/runs/detect/train`. And we can use this weights file to predict new images with the command:

```
yolo predict model=/runs/detect/train2/
weights/best.pt source=/assets/floor1.png
```

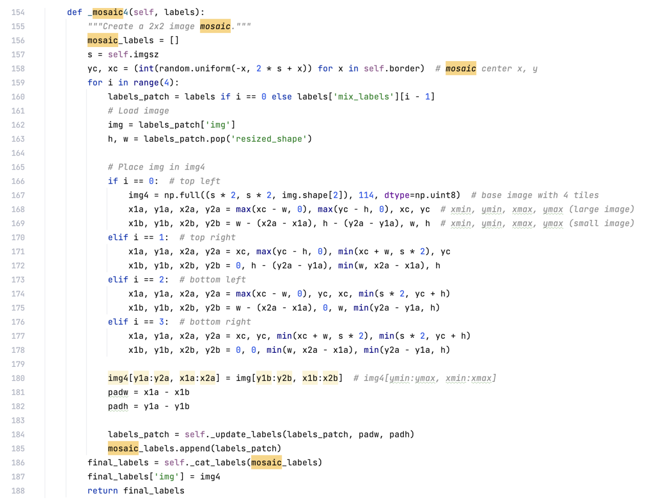
The results of the pictures with bonding boxes are stored in the folder `/runs/detect/predict`.

3.3 *Data Augmentation*

In general, increasing augmentation hyperparameters will reduce and delay overfitting, allowing for longer training and higher final mAP. This time we will mainly focus on three hyperparameters Mosaic, Mixup and Label Smoothing to study how much they can improve the results.

These parameters are stored in the file `default.yaml` in the folder `cfg`. They can be modified directly in the `default.yaml` file, and we need to add the address of this file in the command or add the desired hyperparameters value directly in the command to ensure that these hyperparameters are implemented during training.

3.3.1 Mosaic: One option is to use the Mosaic augmentation technique. This augmentation can combine multiple images (4 or 9) into a single image as the new mosaic image, forcing the model to learn objects in new locations, in partial occlusion, and against different surrounding pixels, which helps prevent overfitting and augment your training data. However, this augmentation is empirically shown to degrade performance if performed through the whole training routine. It is advantageous to turn it off for the last ten training epochs. And this change has been added into the YOLOv8 model. The Mosaic augmentation is defined in the file `/yolo/data/augment.py`, which is shown in Figure 9.



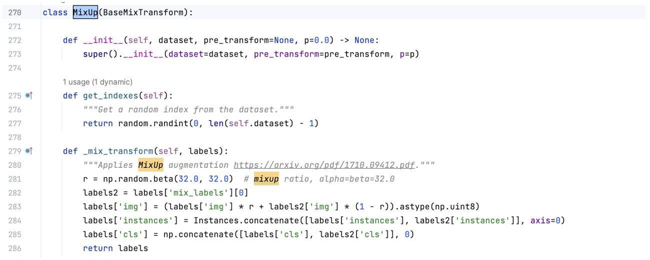
```python
    def _mosaic4(self, labels):
        """Create a 2x2 image mosaic."""
        mosaic_labels = []
        s = self.imgsz
        yc, xc = (int(random.uniform(-x, 2 * s + x)) for x in self.border)  # mosaic center x, y
        for i in range(4):
            labels_patch = labels if i == 0 else labels['mix_labels'][i - 1]
            # Load image
            img = labels_patch['img']
            h, w = labels_patch.pop('resized_shape')

            # Place img in img4
            if i == 0:  # top left
                img4 = np.full((s * 2, s * 2, img.shape[2]), 114, dtype=np.uint8)  # base image with 4 tiles
                x1a, y1a, x2a, y2a = max(xc - w, 0), max(yc - h, 0), xc, yc  # xmin, ymin, xmax, ymax (large image)
                x1b, y1b, x2b, y2b = w - (x2a - x1a), h - (y2a - y1a), w, h  # xmin, ymin, xmax, ymax (small image)
            elif i == 1:  # top right
                x1a, y1a, x2a, y2a = xc, max(yc - h, 0), min(xc + w, s * 2), yc
                x1b, y1b, x2b, y2b = 0, h - (y2a - y1a), min(w, x2a - x1a), h
            elif i == 2:  # bottom left
                x1a, y1a, x2a, y2a = max(xc - w, 0), yc, xc, min(s * 2, yc + h)
                x1b, y1b, x2b, y2b = w - (x2a - x1a), 0, w, min(y2a - y1a, h)
            elif i == 3:  # bottom right
                x1a, y1a, x2a, y2a = xc, yc, min(xc + w, s * 2), min(s * 2, yc + h)
                x1b, y1b, x2b, y2b = 0, 0, min(w, x2a - x1a), min(y2a - y1a, h)

            img4[y1a:y2a, x1a:x2a] = img[y1b:y2b, x1b:x2b]  # img4[ymin:ymax, xmin:xmax]
            padw = x1a - x1b
            padh = y1a - y1b

            labels_patch = self._update_labels(labels_patch, padw, padh)
            mosaic_labels.append(labels_patch)
        final_labels = self._cat_labels(mosaic_labels)
        final_labels['img'] = img4
        return final_labels
```

Fig. 9. Mosaic(4 images)

3.3.2 Mixup: Another option is the Mixup augmentation. It can generate synthetic data by mixing two different images. This involves creating new training samples by linearly interpolating between pairs of existing samples and their corresponding labels, which can improve generalization and robustness. The Mixup augmentation is also defined in the file `/yolo/data/augment.py`, which is shown in Figure 10.

3.3.3 Label Smoothing: Label Smoothing involves modifying the ground truth labels during training to introduce a small amount of uncertainty or noise in the label assignments, as in

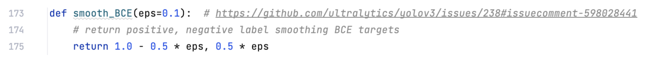
$$Label_{true}^{smooth} = Label_{true} \times (1 - \alpha) + Label_{true} \times \alpha$$



```python
class MixUp(BaseMixTransform):

    def __init__(self, dataset, pre_transform=None, p=0.0) -> None:
        super().__init__(dataset=dataset, pre_transform=pre_transform, p=p)

    1 usage (1 dynamic)
    def get_indexes(self):
        """Get a random index from the dataset."""
        return random.randint(0, len(self.dataset) - 1)

    def _mix_transform(self, labels):
        """Applies MixUp augmentation https://arxiv.org/pdf/1710.09412.pdf."""
        r = np.random.beta(32.0, 32.0)  # mixup ratio, alpha=beta=32.0
        labels2 = labels['mix_labels'][0]
        labels['img'] = (labels['img'] * r + labels2['img'] * (1 - r)).astype(np.uint8)
        labels['instances'] = Instances.concatenate([labels['instances'], labels2['instances']], axis=0)
        labels['cls'] = np.concatenate([labels['cls'], labels2['cls']], 0)
        return labels
```

Fig. 10. Mixup

So it increases the amount of information and provides the relationship between categories in the training data. In summary, it can weaken the certainty of the label compared with one hot label, thereby reducing the possibility of overfitting and enhance the model generalization ability. The Label Smoothing augmentation is defined in the file `/yolo/utils/metrics.py`, which is shown in Figure 11.



```python
def smooth_BCE(eps=0.1):  # https://github.com/ultralytics/yolov3/issues/238#issuecomment-598028441
    # return positive, negative label smoothing BCE targets
    return 1.0 - 0.5 * eps, 0.5 * eps
```

Fig. 11. Label Smoothing

4. HYPERPARAMETER TUNING

Hyperparameter tuning refers to the process of selecting the optimal value for the various parameters and settings and can have a significant impact on the performance of the model, including factors such as accuracy, precision, and recall. In this project, we will try to optimize the result by tuning Mosaic, Mixup, and Label-Smoothing. The dataset we use for tuning contains 1156 training pictures and 80 testing pictures. We run the training with 30 epochs and batch size 0f 8. This is to reduce the complexity of the network. For the final version, we will use a larger dataset with more epochs.

4.1 *Tuning with Mosaic*

We change the value of Mosaic several times and get the result of mAP, which is shown in Table 1.

TABLE I

mAP VALUES UNDER MOSAIC

Mosaic	all classes mAP	dirt
0	0.911	0.786
0.3	0.902	0.767
0.5	0.915	0.785
0.7	0.907	0.99
1	0.914	0.804

From Table 1 we can see that when Mosaic = 0.5, all classes mAP is the highest. However, in this project, we mainly focus on detecting dirt. When Mosaic = 1, the mean average precision is significantly larger than all other options. And ”all classes mAP” is almost the same as Mosaic with 0.5. Therefore, Mosaic = 1 is the best option and will be applied for the next tuning of the parameter.

4.2 *Tuning with Mixup*

With Mosaic being 1, we change the value of Mixup several times and get the result of mAP, which is shown in Table 2 below.

TABLE II
MAP VALUES UNDER MIXUP (MOSAIC=1)

Mixup	all classes mAP	dirt	rulers	scissors
0	0.914	0.804	0.732	0.975
0.1	0.914	0.78	0.771	0.963
0.2	0.906	0.779	0.742	0.976
0.3	0.914	0.8	0.754	0.991
0.4	0.911	0.795	0.74	0.99
0.5	0.914	0.782	0.746	0.981
0.7	0.907	0.783	0.716	0.993
0.9	0.912	0.808	0.716	0.995

From Table 2 we can see that the overall mAP is not improved. However, when Mixup = 0.3, the mAP of some of the items such as rulers and scissors have been significantly improved while the mAP of dirt remains almost the same. Therefore, we decide to use Mixup = 0.3 to have a more precise detection on all the items.

4.3 *Tuning with Label-Smoothing*

With Mosaic being 1, we change the value of Label-Smoothing several times and get the result of mAP, which is shown in Table 3 below.

TABLE III
MAP VALUES UNDER LABEL-SMOOTHING (MOSAIC=1)

Label-Smoothing	all classes mAP	dirt
0	0.914	0.804
0.1	0.914	0.804
0.3	0.914	0.804

From Table 3 we can see that the results not only ”all classes mAP”, but also all other items remain the same under the change of Label-Smoothing. This is because the pictures used for training are already complex enough so that the Label-Smoothing will not bring any effect.

4.4 *Optimal Result from Tuning*

Overall, we can get the most optimized version of tuning, which is Mixup = 0.3 combined with Mosaic = 1. It improves the overall mean average precision from 0.911 to 0.914. Therefore, we can train again with a larger dataset and more epochs using these parameters and get this normalized confusion matrix, which is shown in Figure 12.

4.5 *Comparison*

we compare the results with those that are not tuned. With the same tuning, we compare it again with yolov5 and see the difference between the two systems.

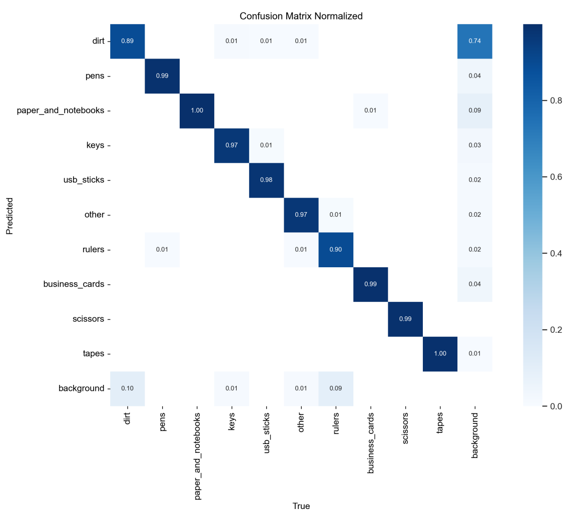


Fig. 12. Confusion Matrix Normalized

4.5.1 Comparison before and after tuning: The dataset we use for the final training contains 9248 training pictures and 3448 testing pictures. We run the training with 100 epochs and batch size 0f 16. The results can be seen in Figure 13 and Figure 14. With Larger datasets and more epochs we can see that mean average precision has been dramatically improved even before tuning. The mAP before tuning is 0.967. After tuning, this value is improved to 0.971. We can see that the precision has already been high enough.

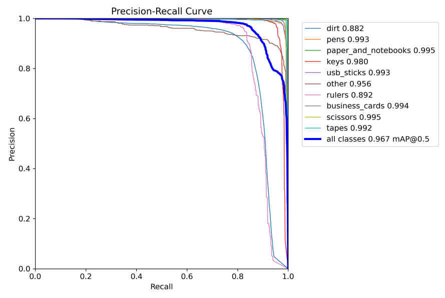


Fig. 13. PR curve before tuning

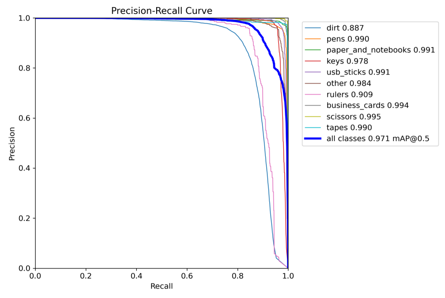
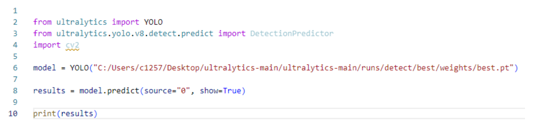


Fig. 14. PR curve after tuning

4.6 *Further Applications: Real-Cam detection*

We use the file “best. pt” from the final training and apply it to real-cam detection. The code for the detection is shown in Figure 15 below.



```python

from ultralytics import YOLO
from ultralytics.yolo.v8.detect.predict import DetectionPredictor
import cv2

model = YOLO("C:/Users/c1257/Desktop/ultralytics-main/ultralytics-main/runs/detect/best/weights/best.pt")

results = model.predict(source="0", show=True)

print(results)
```

Fig. 15. code for real-time detection

After applying this code, we can detect objects in the real world. The precision turns out to be very high with most of the items over mAP of 0.9. Figure 16 below shows the value of the results with an average FPS of around 65 frames per second.

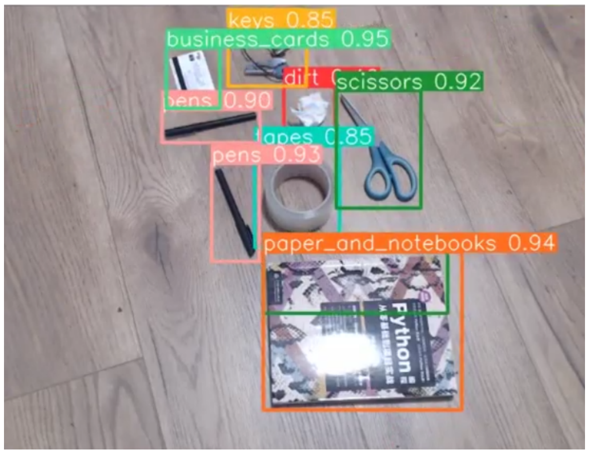


Fig. 16. result of real-time detection

5. SUMMARY AND OUTLOOK

By employing the YOLOv8 object detection model, we have trained our model to accurately identify various types of dirt separate from other items. The performance of the final model reaches mAP for all class of 0.971 and processing speed of around 67 FPS, which is sufficient for the real-time detection task.

Looking ahead, the potential for expansion and further development of this model is enormous. Future research could focus on improving the model’s ability to distinguish between more complex and nuanced categories of objects. This could involve expanding the training dataset to include a wider variety of environments and object types.

We also envisage integrating this model with other robotic systems to create multi-functional robots capable of carrying out a wider array of tasks. For example, we can transplant the model into ROS so that the detection algorithm can cooperate with other actuator of the robots, creating various possibilities the robot system can fulfill.

REFERENCES

[1] Redmon, J., Divvala, S. K., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, Real-Time Object Detection. https://doi.org/10.1109/cvpr.2016.91

[2] Reis, D., Kupec, J., Hong, J., & Daoudi, A. (2023). Real-Time Flying Object Detection with YOLOv8. arXiv preprint arXiv:2305.09972.

[3] Ultralytics. GitHub - ultralytics/ultralytics: NEW - YOLOv8 in PyTorch. GitHub. https://github.com/ultralytics/ultralytics.