Journal of Solid Waste Valorization and Chemical Management, 2026, Vol. 01

Supplementary Files:

DOI: 10.66729/jswvcm202603002

A Dual-Strategy Deep Learning Framework for Automated Defect Detection: Enabling Efficient Echelon Utilization of Spent Lithium-Ion Batteries

Yuhui Peng[1], Yaxin Chi[1,2], Xihua Zhang[1*], Tao Zhang[2], Xuning Zhuang[1], Xiaolong Song[1]

1 School of Resources and Environmental Engineering, Shanghai Polytechnic University; Shanghai Collaborative Innovation Center for WEEE Recycling, Shanghai 201209, China.

2 North Star Advanced Recycling Technology (Qingdao) Co., Ltd., Shandong Qingdao 266109, China.

*Corresponding Author: Xihua Zhang; E-mail: zhangxh@sspu.edu.cn

1. The hybrid detection framework

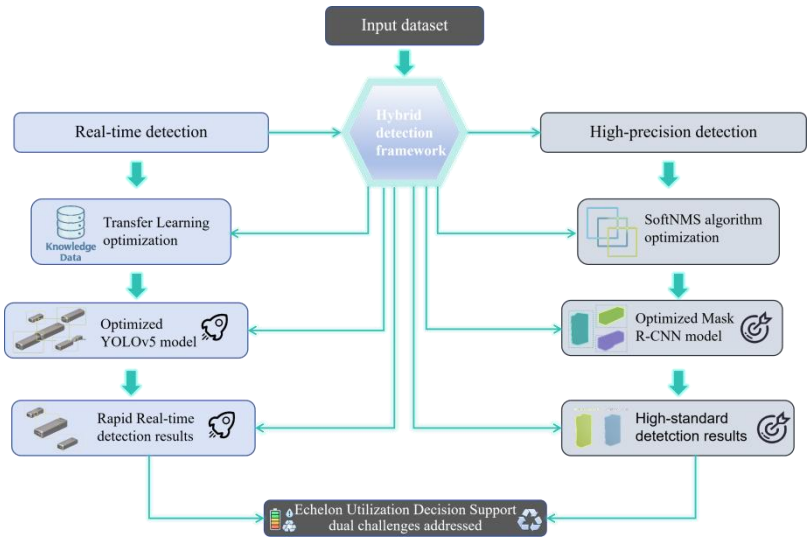

Figure. S1 The hybrid detection framework.

2. Partial images of the original dataset

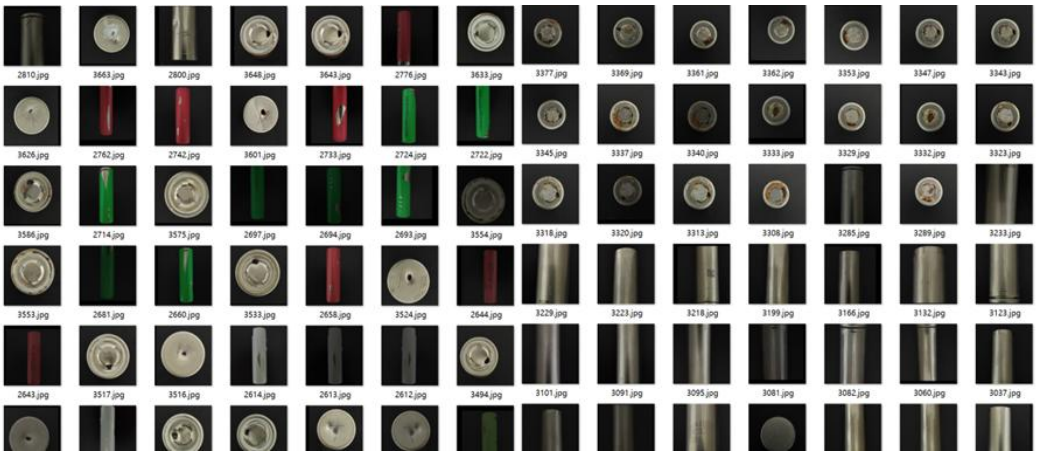

Figure. S2 Partial images of the original dataset.

Received 13 March 2026; Revised 21 April 2026; Accepted 14 May 2026; Available online 28 May 2026.

3. Data labeling and processing

To obtain models with greater generalization ability and robustness while effectively preventing overfitting, a larger training dataset is essential. A sufficient dataset enables better network convergence, thereby improving detection performance during the testing phase. In this work, only 625 defect images were initially collected. Due to the diverse causes of each defect type, their occurrence frequencies vary, making the dataset relatively imbalanced and limited in size. To address this, data augmentation techniques were applied to expand the dataset. Single-sample augmentation methods, such as changing brightness, adding noise, adding random points, translating, and flipping, and expanding the original cylindrical battery image and its annotated generated JSON file. As a result, the dataset was expanded to 3750 images, meeting the requirements for effective model training, as shown in **Figure. S3**.

Figure. S3 Comparison of images before and after data enhancement.

Images were classified based on the type of defect with the highest number of defects in the image, and the specific statistics of the number of defective images are shown in **Table S1.**

Table S1. Number of images of each type of defect before and after data enhancement

Defect Name	Raw data	Data enhanced	After data augmentation
Scratch	85	425	510
Leakage	80	400	480
Stain	129	645	774
Pit	103	515	618
Breakage	228	1140	1368
Total	625	3125	3750

4. Image dataset construction

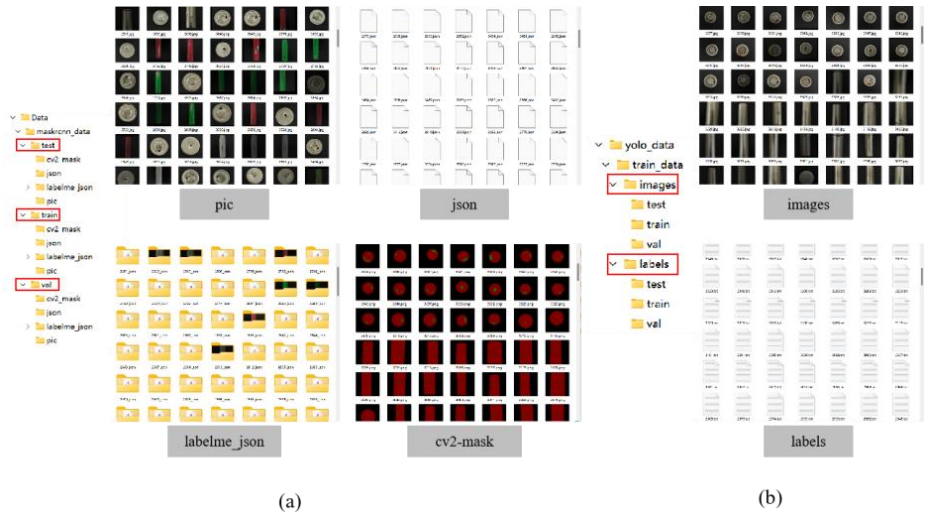

Figure. S4 Structure of datasets corresponding to the models. **(a)** Structure of YOLOv5-7.0 dataset, **(b)** Structure of Mask-RCNN dataset.

5. YOLOv5 model structure and description

The YOLOv5 network model can be classified into five architectures: n, s, m, l and x[1], each differing primarily in network depth and width, which in turn affect the computational cost and model size. To strike a balance between detection performance and computational efficiency, the overall network structure consists of four components: Input, Backbone, Neck(responsible for multi-scale feature fusion), and Head. Its structure is shown in **Figure. S5.**

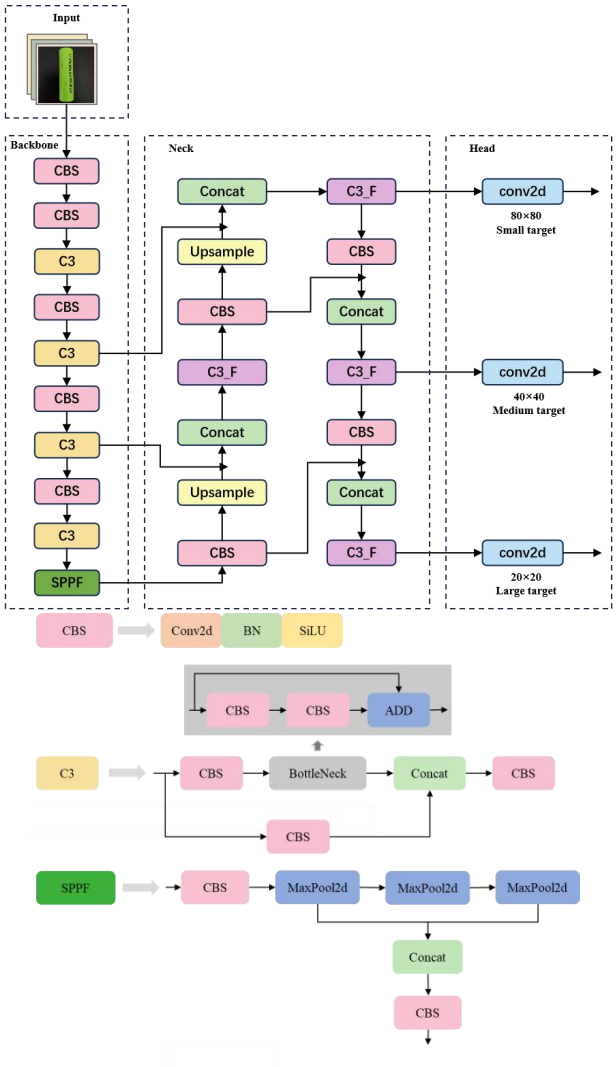

Figure. S5 YOLOv5 overall network structure framework diagram.

6. Flowchart of YOLOv5s model training after introducing transfer learning

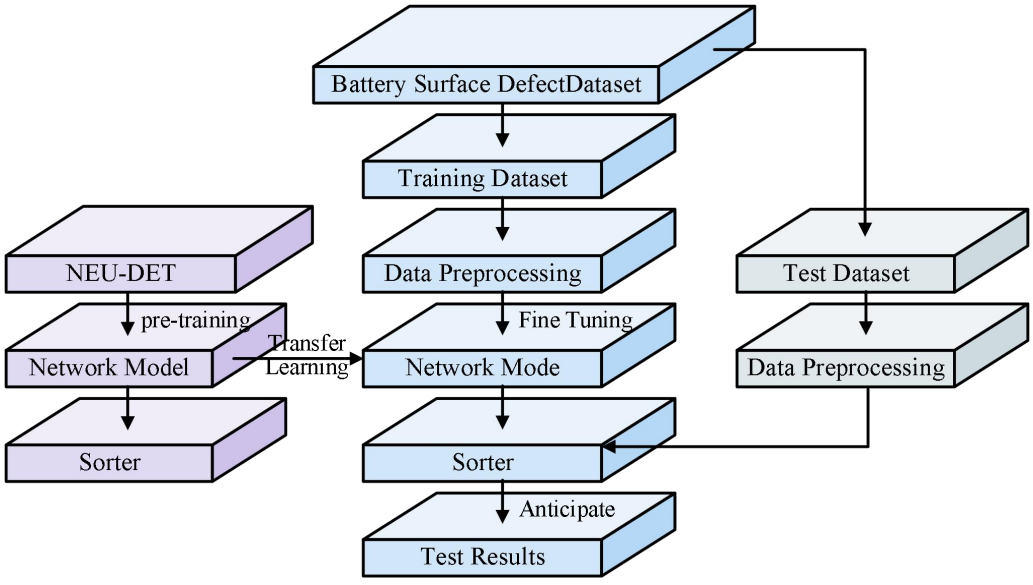

Figure. S6 Flowchart of YOLOv5s model training after introducing transfer learning.

7. Mask R-CNN model structure and description

Mask R-CNN is an extension of the R-CNN series of models belonging to the two-stage network and an improvement on Faster R-CNN [2]. While Faster R-CNN outputs a bounding box and a class label for each proposed region, Mask R-CNN adds a third branch that predicts a segmentation mask for each detected object, enabling instance segmentation. The model architecture consists of four main components: the Backbone, the Region Proposal Network (RPN), Region of Interest Align (RoI Align), and the Output. The overall framework is illustrated in **Figure. S7 and S8.**

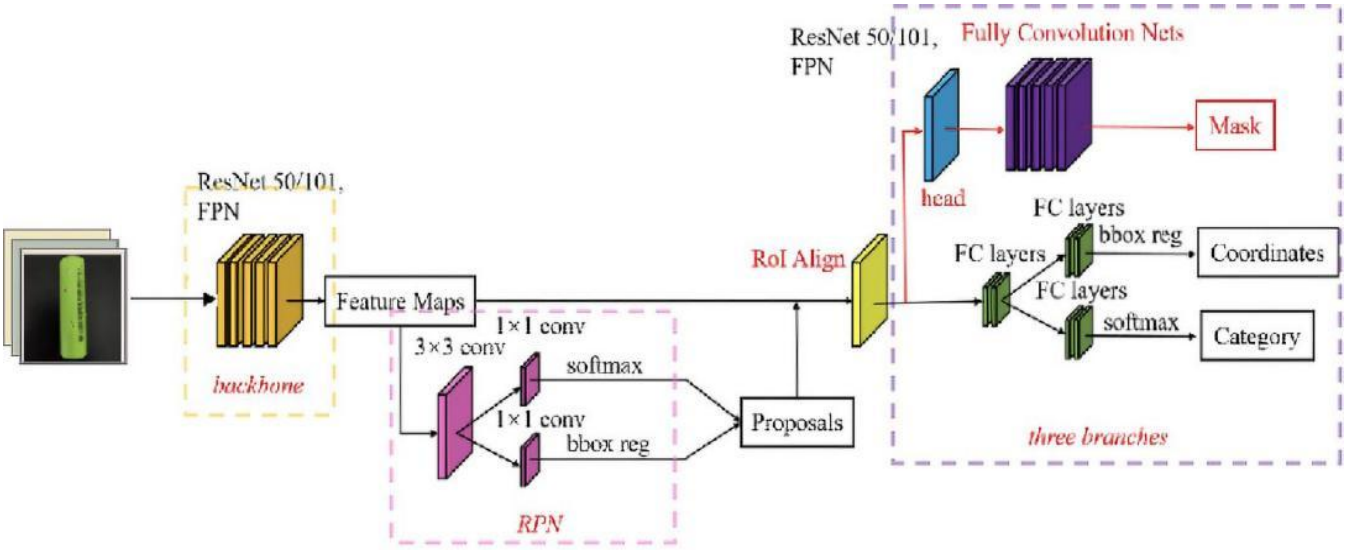

Figure. S7 Mask R-CNN overall network structure framework diagram.

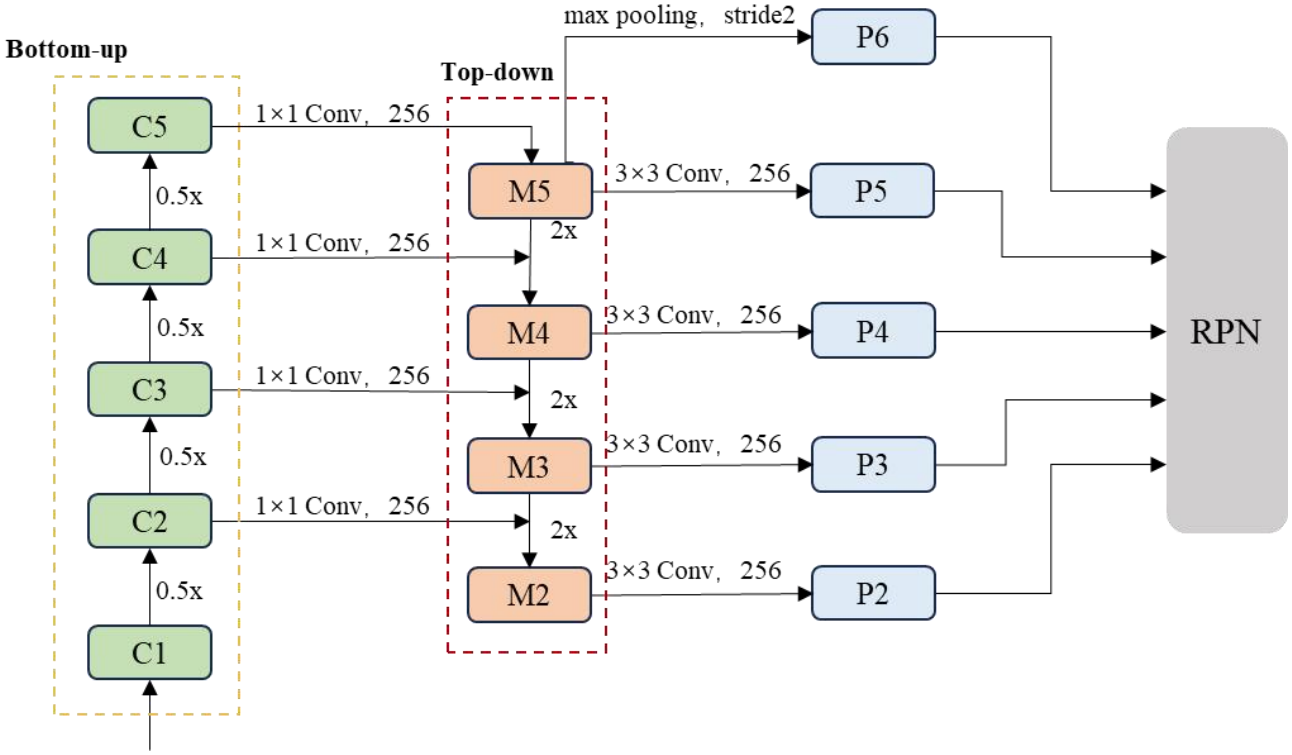

Figure. S8 ResNet50-FPN network architecture.

8. The SoftNMS algorithm replaces the NMS algorithm

The leakage detection phenomenon occurs when small targets, such as scratches, are in a complex background or there is more interference surrounding the defects. In complex backgrounds, overlapping candidate boxes may added together, causing the total overlap to exceed the preset threshold, and the Mask R-CNN might delete these boxes, leading to missed detection. In this work, the model was modified by replacing the NMS algorithm with the SoftNMS algorithm, which substituted minuscule values for the candidate boxes that surpassed the overlap criteria. The formula

was presented in **Eq. (1)**. Experimental validation demonstrated that the improved method significantly reduced missed detection of tiny targets in complex scenarios.

$$S_i = \begin{cases} S_i, iou(m, b_i) \geq N_t \\ S_i(1 - iou(m, b_i)), iou(m, b_i) \geq N_t \end{cases} \quad (1)$$

Where, s_i is the candidate box score, m is the highest confidence box, b_i each candidate box in the output box other than m, $iou\ (m, b_i)$ is the intersection and concurrency ratio between the highest confidence box and each candidate box, N_t and is the NMS threshold i.e. overlap threshold. Its general training flow is shown in **Figure. S9.**

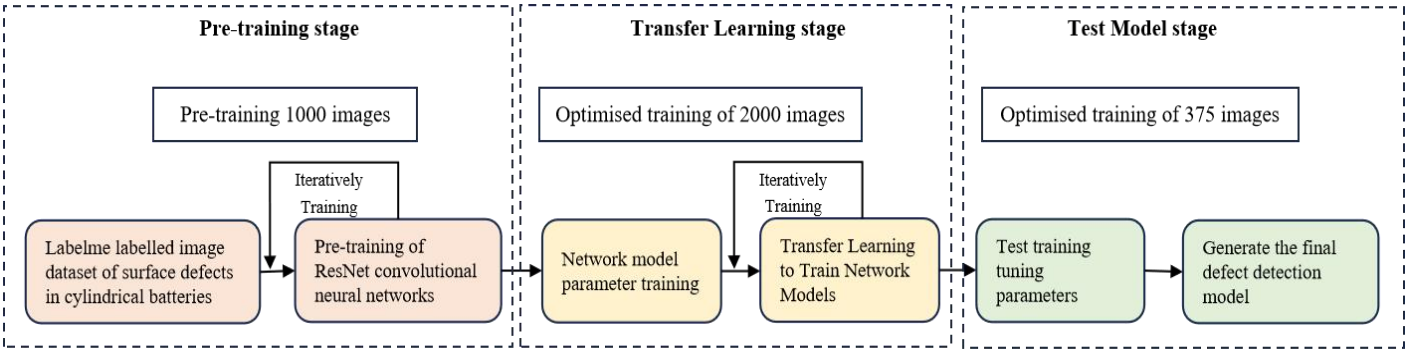

Figure. S9 Mask R-CNN appearance defect detection network model training flowchart.

9. Training configuration details

9.1 YOLOv5 model

Table S2. shows the hardware configuration, software environment, frameworks, and related dependent libraries used to train the YOLOv5 model.

Table S2. Software and hardware environment

Environment	Version or parameter
CPU	Intel(R) Xeon(R) Gold 6330
GPU	RTX3090 24 G
System	Ubuntu20.04
Languages	Python3.7
Framework	PyTorch1.13.1
Dependency libraries	OpenCV4.8、numpy

After configuring the hardware and software environment for network training, it is also essential to adjust relevant parameters, including both model parameters and hyperparameters, as detailed in **Table S3.**

(1) Model parameters mainly include the structural elements such as convolutional kernel size, number of convolutional layers, and pooling layers. These are typically fixed but can be fine-tuned depending on task requirements.

(2) Hyperparameters include the learning rate, batch size, number of iterations, etc., in which the size of the training batch (Batchsize) should be optimally matched with the GPU's operating memory. In this work, it was set to 8. The number of iterations (Epoch), which affects convergence of the loss function and mean average precision (mAP), was set to 300.

Table S3. YOLOv5s model hyperparameters

Parameter	Numerical value
Learning rate	0.01
Weight decay	0.0005
Momentum	0.937
Batch size	8
Epoch	300

9.2 Mask R-CNN model

Table S4. shows the hardware configuration, software environment, frameworks, and related dependent libraries used for Mask R-CNN model training.

Table S4. Hardware and software environment

Environment	Version or parameter
CPU	15 vCPU Intel(R) Xeon(R) Platinum 8358P
GPU	RTX3090 24GB
System	Ubuntu20.04
Languages	Python3.8
Framework	tensorflow-gpu2.5.0
Dependency libraries	h5py、keras2.5.0、OpenCV4.7、numpy、scipy

After configuring the hardware and software environment for model training, the initial parameters are set based on backbone weights pre-trained on the ImageNet dataset, which helps shorten the training cycle. Detailed training parameter settings are provided in **Table S5**.

Table S5. Initial hyperparameters for Mask R-CNN model training

Parameters	Numerical value
Epoch	50
Batch size	1
Learning rate	0.001
Weight decay	0.0001
NMS threshold value	0.7
Warm-up factor	0.001
Warm-up steps	1000

In this work, we set the iterative training 50 times, due to the limited memory, this study set the batch size to 1, and the initial learning rate was set to 0.001; in order to ensure the stability of the model learning, this experiment adopted the warm-up strategy, with a warm-up factor of 0.001, and the number of warm-up steps was 1000.

9.3 Evaluation indicators

To evaluate the effectiveness of a network model, a set of evaluation indicators was employed. Specifically, five indicators were used in this work: accuracy, precision, recall, mean average precision (mAP), and frames per second (FPS) (Lawal, 2023; Wang et al., 2023). These evaluation indicators comprehensively reflected detection accuracy, missed detection, and real-time performance. The parameters involved in these indicators can be represented by a confusion matrix, as shown in **Table S6**, TP indicates the number of real defective targets that were correctly identified by the network, TN indicates the number of non-targeted defects that were not detected by the network, FP indicates the number of non-targeted defects that were incorrectly identified by the network as actual target defects, and FN indicates the number of actual target defects that were not identified by the network.

Table S6. Confusion matrix

Real/projected value	actual positive	actual negative
predicted positive	TP	FP
predicted negative	FN	TN

Accuracy is defined as the proportion of correctly predicted samples among all samples. Its calculated formula is shown in **Eq. (2).**

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{2}$$

Precision is defined as a ratio of correctly identified defective targets to the number of identified defective targets. A higher precision value indicates fewer false detection. Its calculation formula is shown in **Eq. (3).**

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

Recall plays an important role in industrial detection. It is defined as the ratio of correctly detected actual targets to the total number of actual targets. A higher recall value indicates a lower likelihood of missed detection. Its calculation formula is shown in **Eq. (4).**

$$Recall = \frac{TP}{TP + FN} \tag{4}$$

The average precision (AP) mean value is calculated by averaging the average precision (AP) values across all categories of defective targets. The AP reflects the model's detection performance for each defect type and balances precision and recall. The precision rate (P) is taken as the horizontal coordinate and the recall rate (R) as the vertical coordinate, thereby obtaining the precision-recall curve, namely the P-R curve. The AP is defined as the area under the P-R curve, which can be calculated by integration. In contrast, the mean average precision (mAP) is the cumulative average of the AP values of all defective categories (the number of categories is n). Its calculation formula is shown in **Eq. (5)**.

$$\mathrm{m}AP = \frac{1}{N}\sum_{i=1}^{N} AP_i \tag{5}$$

In the defect detection networks, the frames per second (FPS) is the number of images detected per second. Alternatively, it can be expressed as the refresh frequency of an image or the number of frames per second.

10. Partial source code of cylindrical stepped battery appearance defect detection method based on Mask R-CNN

```
ROOT_DIR = os.path.abspath(r"    ")
sys.path.append(ROOT_DIR)    # To find local version of the library
from mrcnn import utils
from mrcnn import visualize
from mrcnn import model as modellib
from mrcnn.config import Config
from mrcnn.model import log
from mrcnn.visualize import display_images
from PIL import Image
MODEL_DIR = os.path.join(ROOT_DIR, "logs")
COCO_WEIGHT_PATH = os.path.join(ROOT_DIR, "mask_rcnn_coco.h5")
if not os.path.exists(COCO_WEIGHT_PATH):
    utils.download_trained_weights(COCO_WEIGHT_PATH)
class BatteryConfig(Config):
      NAME = "battery"
      GPU_COUNT = 1
IMAGES_PER_GPU = 1
# background + 6 defects      Stain/Pit/Breakage/Scratch/Weeping/battery
NUM_CLASSES = 1 + 6
      IMAGE_MIN_DIM = 640
      IMAGE_MAX_DIM = 640
      RPN_ANCHOR_SCALES = (8 * 6, 16 * 6, 32 * 6, 64 * 6, 128 * 6)
      TRAIN_ROIS_PER_IMAGE = 100
      STEPS_PER_EPOCH = 100
      VALIDATION_STEPS = 50
config = BatteryConfig()
config.display()
class BatteryDataset(utils.Dataset):
      def get_obj_index(self, image):
            n = np.max(image)
```

```
        return n
    def from_yaml_get_class(self, image_id):
        info = self.image_info[image_id]
        with open(info['yaml_path']) as f:
            temp = yaml.load(f.read(), Loader=yaml.FullLoader)
            labels = temp['label_names']
            del labels[0]
        return labels
    def draw_mask(self, num_obj, mask, image, image_id):
        info = self.image_info[image_id]
        for index in range(num_obj):
            for i in range(info['width']):
                for j in range(info['height']):
                    at_pixel = image.getpixel((i, j))
                    if at_pixel == index + 1:
                        mask[j, i, index] = 1
        return mask
    def load_battery(self, count, image_folder, mask_folder, imglist, dataset_root_path):
        self.add_class("battery", 1, "Scratch")
        self.add_class("battery", 2, "Pit")
        self.add_class("battery", 3, "Breakage")
        self.add_class("battery", 4, "Weeping")
        self.add_class("battery", 5, "Stain")
        self.add_class("battery", 6, "Battery")
        for i in range(count):
            filestr = imglist[i].split(".")[0]
            mask_path = mask_folder + "/" + filestr + ".png"
            yaml_path = dataset_root_path + "/labelme_json/" + filestr + "_json/info.yaml"
            print(dataset_root_path + "/labelme_json/" + filestr + "_json/img.png")
            cv_img = cv2.imread(dataset_root_path + "/labelme_json/" + filestr + "_json/img.png")
            self.add_image("battery", image_id=i, path=image_folder + "/" + imglist[i],
                           width=cv_img.shape[1],
height=cv_img.shape[0],
```

```
mask_path=mask_path, yaml_path=yaml_path)
    def load_mask(self, image_id):
        global iter_num
        print("image_id", image_id)
        info = self.image_info[image_id]
        count = 1
        img = Image.open(info['mask_path'])
        num_obj = self.get_obj_index(img)
        mask = np.zeros([info['height'], info['width'], num_obj], dtype=np.uint8)
        mask = self.draw_mask(num_obj, mask, img, image_id)
        occlusion = np.logical_not(mask[:, :, -1]).astype(np.uint8)
        for i in range(count - 2, -1, -1):
            mask[:, :, i] = mask[:, :, i] * occlusion
            occlusion = np.logical_and(occlusion, np.logical_not(mask[:, :, i]))
        labels = []
        labels = self.from_yaml_get_class(image_id)
        labels_form = []
        for i in range(len(labels)):
            if labels[i].find("Scratch") != -1:
                labels_form.append("Scratch")
            if labels[i].find("Pit") != -1:
                labels_form.append("Pit")
            if labels[i].find("Breakage") != -1:
                labels_form.append("Breakage")
            if labels[i].find("Weeping") != -1:
                labels_form.append("Weeping")
            if labels[i].find("Stain") != -1:
                labels_form.append("Stain")
            if labels[i].find("Battery") != -1:
                labels_form.append("Battery")
        class_ids = np.array([self.class_names.index(s) for s in labels_form])
        return mask, class_ids.astype(np.int32)
def get_ax(rows=1, cols=1, size=8):
```

```
    _, ax = plt.subplots(rows, cols, figsize=(size * cols, size * rows))
return ax
```

11. 11. Partial source code of cylindrical stepped battery appearance defect detection method based on YOLOv5

```
def train(hyp, opt, device, callbacks):
    save_dir, epochs, batch_size, weights, single_cls, evolve, data, cfg, resume, noval, nosave, workers, freeze,
mask_ratio = \
        Path(opt.save_dir), opt.epochs, opt.batch_size, opt.weights, opt.single_cls, opt.evolve, opt.data,
opt.cfg, \
        opt.resume, opt.noval, opt.nosave, opt.workers, opt.freeze, opt.mask_ratio
    # Directories
    w = save_dir / 'weights'
    (w.parent if evolve else w).mkdir(parents=True, exist_ok=True)
    # Hyperparameters
    if isinstance(hyp, str):
        with open(hyp, errors='ignore') as f:
            hyp = yaml.safe_load(f)
    LOGGER.info(colorstr('hyperparameters: ') + ', '.join(f'{k}={v}' for k, v in hyp.items()))
    opt.hyp = hyp.copy()
    # Save run settings
    if not evolve:
        yaml_save(save_dir / 'hyp.yaml', hyp)
        yaml_save(save_dir / 'opt.yaml', vars(opt))
    # Loggers
    data_dict = None
    if RANK in {-1, 0}:
        logger = GenericLogger(opt=opt, console_logger=LOGGER)
    # Config
    plots = not evolve and not opt.noplots
    overlap = not opt.no_overlap
    cuda = device.type != 'cpu'
    init_seeds(opt.seed + 1 + RANK, deterministic=True)
    with torch_distributed_zero_first(LOCAL_RANK):
```

```
        data_dict = data_dict or check_dataset(data)
    train_path, val_path = data_dict['train'], data_dict['val']
    nc = 1 if single_cls else int(data_dict['nc'])
    names = {0: 'item'} if single_cls and len(data_dict['names']) != 1 else data_dict['names']
    is_coco = isinstance(val_path, str) and val_path.endswith('coco/val2017.txt')  # COCO dataset
    # Model
    check_suffix(weights, '.pt')
    pretrained = weights.endswith('.pt')
    if pretrained:
        with torch_distributed_zero_first(LOCAL_RANK):
            weights = attempt_download(weights)
        ckpt = torch.load(weights, map_location='cpu')
        model = SegmentationModel(cfg or ckpt['model'].yaml, ch=3, nc=nc,
anchors=hyp.get('anchors')).to(device)
        exclude = ['anchor'] if (cfg or hyp.get('anchors')) and not resume else []
        csd = ckpt['model'].float().state_dict()
        csd = intersect_dicts(csd, model.state_dict(), exclude=exclude)
        model.load_state_dict(csd, strict=False)
        LOGGER.info(f'Transferred {len(csd)}/{len(model.state_dict())} items from {weights}')
    else:
        model = SegmentationModel(cfg, ch=3, nc=nc, anchors=hyp.get('anchors')).to(device)
    amp = check_amp(model)
    # Freeze
    freeze = [f'model.{x}.' for x in (freeze if len(freeze) > 1 else range(freeze[0]))]
    for k, v in model.named_parameters():
        v.requires_grad = True
        if any(x in k for x in freeze):
            LOGGER.info(f'freezing {k}')
            v.requires_grad = False
    # Image size
    gs = max(int(model.stride.max()), 32)
    imgsz = check_img_size(opt.imgsz, gs, floor=gs * 2)
    # Batch size
```

```
    if RANK == -1 and batch_size == -1:
        batch_size = check_train_batch_size(model, imgsz, amp)
        logger.update_params({'batch_size': batch_size})
    # Optimizer
    nbs = 64
    accumulate = max(round(nbs / batch_size), 1)
    hyp['weight_decay'] *= batch_size * accumulate / nbs
    optimizer = smart_optimizer(model, opt.optimizer, hyp['lr0'], hyp['momentum'], hyp['weight_decay'])
    # Scheduler
    if opt.cos_lr:
        lf = one_cycle(1, hyp['lrf'], epochs)
    else:
        lf = lambda x: (1 - x / epochs) * (1.0 - hyp['lrf']) + hyp['lrf']
    scheduler = lr_scheduler.LambdaLR(optimizer, lr_lambda=lf)
    # EMA
    ema = ModelEMA(model) if RANK in {-1, 0} else None
    # Resume
    best_fitness, start_epoch = 0.0, 0
    if pretrained:
        if resume:
            best_fitness, start_epoch, epochs = smart_resume(ckpt, optimizer, ema, weights, epochs, resume)
        del ckpt, csd
    # DP mode
    if cuda and RANK == -1 and torch.cuda.device_count() > 1:
        LOGGER.warning(
            'WARNING DP not recommended, use torch.distributed.run for best DDP Multi-GPU results.\n'
            'See Multi-GPU Tutorial at https://docs.ultralytics.com/yolov5/tutorials/multi_gpu_training to
get started.'
        )
        model = torch.nn.DataParallel(model)
    # SyncBatchNorm
    if opt.sync_bn and cuda and RANK != -1:
        model = torch.nn.SyncBatchNorm.convert_sync_batchnorm(model).to(device)
```

```
LOGGER.info('Using SyncBatchNorm()')
```

References

[1] Zhang, C.; Hu, X.; Wei, S.; *et al.* Remote sensing object detection based on deep learning. *Computer Engineering and Design*. 2024, 45, (02), 594-600.

[2] He, K.; Gkioxari, G.; Dollar, P.; *et al.* Mask R-CNN. *IEEE transactions on pattern analysis and machine intelligence.* 2020, 42, (2), 386-397.