NYCU CV2025 HW3

GitHub Link:
https://github.com/CEJiang/NYCU-Computer-Vision-2025-Spring-HW3

1. Introduction

In this assignment, we developed a robust cell instance segmentation model capable of accurately classifying and segmenting cells in microscopy images from the HW3 dataset. To enhance the model's generalization ability, we adopted the Mask R-CNN architecture with Swin-Tiny [2] as the backbone, a Region Proposal Network (RPN) [7], a Path Aggregation Network (PAN) [9], and dedicated classification and segmentation heads.

Given the limited size of the dataset, we applied Repeated Augmentation [1] during training to effectively increase the diversity of training samples.

To further boost inference performance, we introduced multiple Test-Time Augmentation (TTA) strategies [10], including horizontal and vertical flips, 90°/180° rotations, brightness and contrast adjustments, hue shifts, Gaussian blur, and transposition. Each augmented image was transformed back to the original space and aggregated, followed by Non-Maximum Suppression (NMS) to remove redundant predictions and retain the most confident ones.

Our best configuration achieved a mAP of 0.4308, demonstrating the overall effectiveness of our model architecture, training strategy, and inference-time enhancements.

2. Method

2.0 Data split

Since no separate validation set was provided for this task, we manually split the training set into a training and validation subset with an 80:20 ratio. This split was used for hyperparameter tuning.

2.1 Data Preprocessing

HorizontalFlip	p=0.3
VerticalFlip	p=0.4
RandomRotate90	p=0.4
ElasticTransform	alpha=1 sigma=50 p=0.3
GridDistortion	num_steps=5 distort_limit=0.3 p=0.3
OpticalDistortion	distort_limit=0.05 p=0.2
ColorJitter	brightness=0.2 contrast=0.2 saturation=0.2 hue=0.1 p=0.3
GaussNoise	p=0.2
Normalize	Mean = [0.485, 0.456, 0.406] Std = [0.229, 0.224, 0.225]
ToTensorV2	

1. HorizontalFlip

   Cells under the microscope do not have a fixed orientation, and their left-right symmetry is often biologically irrelevant. To increase data diversity and model generalization, we apply horizontal flipping as a form of geometric augmentation. This transformation randomly mirrors the image along the vertical

axis with a probability of 30%, allowing the model to learn robust features regardless of the horizontal orientation of the cells.

2. VerticalFlip

   Like horizontal flipping, vertical flipping accounts for the arbitrary positioning of cells in microscopy slides. By flipping the image along the horizontal axis with a probability of 40%, we simulate variations in cell alignment caused by random placement or imaging angles. This augmentation enhances the model's ability to generalize to diverse cell orientations in real-world conditions.

3. RandomRotate90

   Cells in microscopic images may appear at any rotation due to slide orientation or sample preparation. We apply RandomRotate90, which rotates the image by 90°, 180°, or 270° at random. This augmentation, applied with a 40% probability, helps the model become invariant to rotational variations and ensures consistent performance across diverse imaging conditions.

4. ElasticTransfrom

   Cells often exhibit natural deformations such as stretching and distortion under a microscope. In our augmentation strategy, we apply ElasticTransform with alpha=1 to simulate subtle yet realistic shape changes, avoiding overly unnatural distortions. The parameter sigma=50 ensures that the deformations occur over larger, smoother regions, mimicking global shape variations caused by cell movement or tension on the slide. This helps the model learn to segment cells accurately even when their shapes are irregular. Additionally, this augmentation is applied with a probability of 30%.

5. GridDistortion

   Due to imaging artifacts or slide pressure, cells may appear locally distorted under the microscope. To simulate such grid-like spatial distortions, we apply GridDistortion with num_steps=5 and distort_limit=0.3. This augmentation divides the image into a 5×5 grid and allows each grid point to be randomly displaced up to 30%, generating smooth and realistic deformations. We apply this transformation with a probability of 30% to improve the model's robustness against local distortions in real-world scenarios.

6. OpticalDistortion

   Optical distortion can occur at the edges of microscopy images due to lens characteristics, leading to barrel or pincushion effects. To simulate such phenomena, we use OpticalDistortion with distort_limit=0.05, allowing up to ±5% spatial distortion. This augmentation mimics lens-induced curvature, especially around the image edges. We apply it with a 20% probability to help the model adapt to optical deformations and improve segmentation accuracy in real-world imaging conditions.

7. ColorJitter

   To improve the model's robustness to color variation caused by staining procedures or imaging hardware, we apply ColorJitter as a data augmentation technique. This transform randomly adjusts: brightness (±20%), contrast (±20%), saturation (±20%), hue (±10%). These perturbations simulate real-world color variability in cell images. The transform is applied with a probability of 0.3 to ensure that the model generalizes well under diverse appearance conditions.

8. GaussNoise

   Microscopic images are often affected by sensor noise or low-light conditions, resulting in random artifacts in the image. To simulate such real-world interference and improve the model's robustness, we apply GaussNoise, which adds Gaussian-distributed random noise to the image. This helps the model learn to ignore noise and focus on the true shape and boundaries of the cells. We apply this augmentation with a probability of 20%.

9. Normalize

   All inputs were normalized with the ImageNet mean and std for pretrained compatibility.

10. ToTensorV2

    Converts a CV2 image (in [H, W, C] format with uint8 pixel values) to a PyTorch tensor in [C, H, W] format, scaling pixel values to the [0.0, 1.0] range.

2.2 Model Architecture

```python
class PANet(nn.Module):
    def __init__(self, channels):
        super().__init__()
        c2, c3, c4, c5 = channels

        # Top-down
        self.reduce_c5 = nn.Conv2d(c5, c4, 1)
        self.reduce_c4 = nn.Conv2d(c4, c3, 1)
        self.reduce_c3 = nn.Conv2d(c3, c2, 1)

        self.up_c4 = nn.Conv2d(c4 + c4, c4, 3, padding=1)
        self.up_c3 = nn.Conv2d(c3 + c3, c3, 3, padding=1)
        self.up_c2 = nn.Conv2d(c2 + c2, c2, 3, padding=1)

        # Bottom-up
        self.down_c3 = nn.Conv2d(c2, c3, 3, stride=2, padding=1)
        self.fuse_c3 = nn.Conv2d(c3 + c3, c3, 3, padding=1)

        self.down_c4 = nn.Conv2d(c3, c4, 3, stride=2, padding=1)
        self.fuse_c4 = nn.Conv2d(c4 + c4, c4, 3, padding=1)

        self.down_c5 = nn.Conv2d(c4, c5, 3, stride=2, padding=1)
        self.fuse_c5 = nn.Conv2d(c5 + c5, c5, 3, padding=1)

    def forward(self, feats):
        c2, c3, c4, c5 = feats['0'], feats['1'], feats['2'], feats['3']

        # Top-down
        p5 = self.reduce_c5(c5)
        p4 = self.up_c4(torch.cat([F.interpolate(p5, scale_factor=2, mode='nearest'), c4], dim=1))
        p4_reduced = self.reduce_c4(p4)
        p3 = self.up_c3(torch.cat([F.interpolate(p4_reduced, scale_factor=2, mode='nearest'), c3], dim=1))
        p3_reduced = self.reduce_c3(p3)
        p2 = self.up_c2(torch.cat([F.interpolate(p3_reduced, scale_factor=2, mode='nearest'), c2], dim=1))

        # Bottom-up
        p3_out = self.fuse_c3(torch.cat([self.down_c3(p2), p3], dim=1))
        p4_out = self.fuse_c4(torch.cat([self.down_c4(p3_out), p4], dim=1))
        p5_out = self.fuse_c5(torch.cat([self.down_c5(p4_out), p5], dim=1))

        return {
            '0': p2,
            '1': p3_out,
            '2': p4_out,
            '3': p5_out
        }
```

```python
class SwinBackboneWithFPN(nn.Module):
    def __init__(self):
        super().__init__()
        self.backbone = swin_v2_t(weights=Swin_V2_T_Weights.DEFAULT).features
        # print(self.backbone)
        self.body = IntermediateLayerGetter(
            self.backbone,
            OrderedDict({
                '1': '0',
                '3': '1',
                '5': '2',
                '7': '3'
            }))

        self.out_channels = 256

        self.fpn = FeaturePyramidNetwork(
            in_channels_list=[96, 192, 384, 768],
            # in_channels_list=[128, 256, 512, 1024],
            # in_channels_list=[192, 384, 768, 1536],
            out_channels=self.out_channels,
            extra_blocks=LastLevelMaxPool()
        )

    def forward(self, x):
        x = self.body(x)

        x = {k: v.permute(0, 3, 1, 2) for k, v in x.items()}  # [B, C, H, W]

        x = self.fpn(x)

        return x
```

```python
class BackboneWithFPNAndPAN(nn.Module):
    def __init__(self, backbone_fpn, panet):
        super().__init__()
        self.backbone_fpn = backbone_fpn
        self.panet = panet
        self.out_channels = 256

    def forward(self, x):
        fpn_feats = self.backbone_fpn(x)
        pan_feats = self.panet(fpn_feats)
        return pan_feats
```

```python
def swin_model_fpn(num_classes, device, args):
    swin_backbone = SwinBackboneWithFPN()
    panet = PANet([swin_backbone.out_channels] * 4)
    swin_backbone_panet = BackboneWithFPNAndPAN(swin_backbone, panet)


    anchor_sizes = ((4,), (8,), (16,), (32,), (64,), (128,), (256))
    anchor_aspect_ratios = ((0.5, 1.0, 2.0),) * 4

    anchor_generator = AnchorGenerator(
        sizes=anchor_sizes,
        aspect_ratios=anchor_aspect_ratios
    )
    num_anchors = anchor_generator.num_anchors_per_location()[0]

    roi_pooler = MultiScaleRoIAlign(
        featmap_names=['0', '1', '2', '3'],
        output_size=7,
        sampling_ratio=2
    )

    mask_roi_pooler = MultiScaleRoIAlign(
        featmap_names=['0', '1', '2', '3'],
        output_size=14,
        sampling_ratio=2
    )

    model = MaskRCNN(
        backbone=swin_backbone_panet,
        rpn_anchor_generator=anchor_generator,
        box_roi_pool=roi_pooler,
        mask_roi_pool=mask_roi_pooler,
        num_classes=num_classes
    )

    model.roi_heads.positive_fraction = 0.25

    in_features = model.roi_heads.box_predictor.cls_score.in_features
    model.rpn.head = RPNHead(in_channels=256, num_anchors=num_anchors)
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes=num_classes)

    in_features_mask = model.roi_heads.mask_predictor.conv5_mask.in_channels
    hidden_layer = 256
    model.roi_heads.mask_predictor = MaskRCNNPredictor(
        in_features_mask, hidden_layer, num_classes=num_classes
    )

    return model.to(device)
```

Backbone: Swin-Tiny

Pre-trained Weights: Swin_V2_T_Weights.DEFAULT

We adopt Swin-Tiny as the backbone with pre-trained weights, and integrate it with a Feature Pyramid Network (FPN) as the neck. Since FPN requires multi-scale feature maps, we utilize IntermediateLayerGetter to extract outputs from specific stages in Swin-Tiny. The architecture of Swin-Tiny consists of:

- Linear embedding,
- A sequence of Swin Transformer Blocks followed by patch merging,

– Repeated for four stages.

We extract features from the outputs of the 1st, 3rd, 5th, and 7th blocks, which correspond to increasing receptive fields and match the 4 levels (0–3) required by the FPN.

Neck:

Anchor_sizes = ((4,), (8,), (16,), (32,), (64,), (128,), (256))

Anchor_aspect_ratios = ((0.5, 1.0, 2.0),) * 4

To improve the model's performance in multi-scale object detection tasks, we integrated the Feature Pyramid Network (FPN) and Path Aggregation Network (PAN) into the neck of our architecture. FPN builds a top-down semantic hierarchy by fusing high-level semantic information with low-level details. However, relying solely on FPN may not be sufficient to preserve fine-grained localization cues. To address this, we introduced PAN as a complementary structure. Its bottom-up path effectively enhances spatial detail propagation, improving localization accuracy, particularly for small object detection.

Based on statistical analysis of mask sizes in the training dataset, we selected (8, 16, 32, 64) as the core anchor sizes, and additionally included 4-, 128- and 256-pixel anchors to extend the detection capability for extremely small and large objects. The aspect ratios were set to (0.5, 1.0, 2.0), covering wide, square, and elongated shapes in accordance with the object shape distribution in our dataset. These anchors were applied to four feature maps enhanced by both FPN and PAN, enabling the Region Proposal Network (RPN) to generate proposals that are adaptive to both scale and shape.

Our customized PAN module first performs a top-down refinement, progressively upsampling high-level features (from C5 to C2) and fusing them with the corresponding lateral features through concatenation and convolution. Then, a bottom-up path starts from the lowest-level feature map (P2), performing multiple layers of downsampling and fusion to reinforce feature consistency and localization strength. This dual-path design enables the final multi-scale feature representations to retain both semantic richness and precise spatial detail, significantly improving

detection and segmentation performance for small and complex-shaped objects.

Head:

For the head components, we follow the standard Mask R-CNN design, which includes two major heads: a box head for object classification and bounding box regression, and a mask head for instance segmentation.

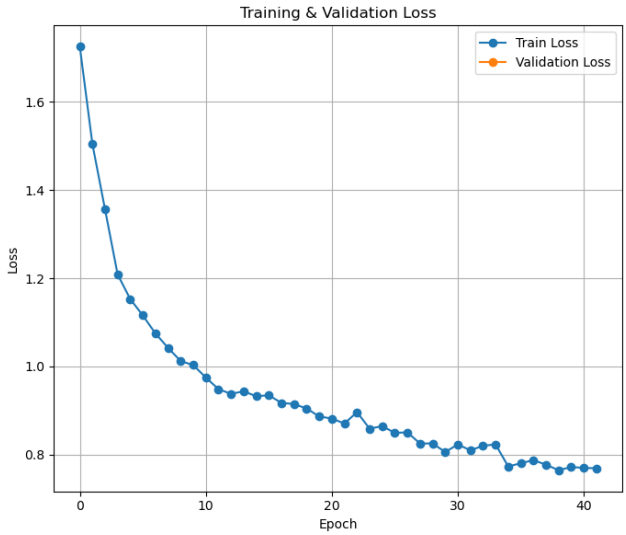
The box head is implemented using FastRCNNPredictor, which takes the ROI-pooled features and outputs classification scores and refined bounding boxes. The mask head uses MaskRCNNPredictor, which predicts a binary segmentation mask for each ROI. Both heads are adjusted to support our target number of classes.

Since our FPN output channels are fixed to 256, the box head input feature dimension and mask head hidden layer size are also set to 256 to maintain consistency.

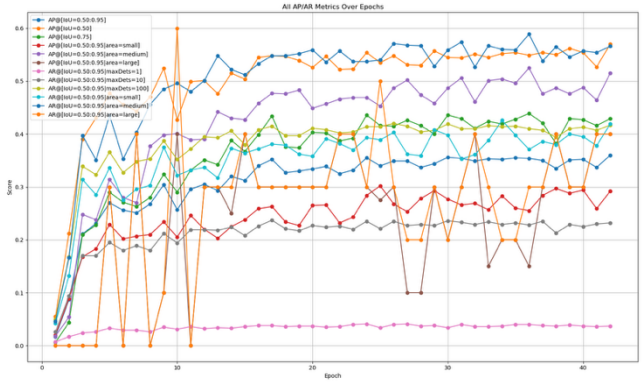
2.3 Hyperparameters

Pretrained Weight	Swin_V2_T_Weights
Learning rate	1×10^{-4}
Batch size	2
Epochs	30
decay	5×10^{-3}
Optimizer	AdamW
Eta_min	1×10^{-6}
T_max	30
Scheduler	CosineAnnealing
Criterion	CrossEntropyLoss (Classification) Smooth L1 Loss (Localization) Binary Cross Entropy Loss (Mask)

3. Results



The training loss decreases steadily over 40 epochs, dropping from ~1.7 to around 0.75. The curve indicates consistent convergence with no signs of overfitting.



- AP@[0.50:0.95] and AP@0.50 reach above 0.55–0.60, indicating good detection performance.
- AP_small and AR_small show a clear advantage due to Swin + FPN + PAN design, which is expected for small-object segmentation.
- While AP_medium is also stable (~0.45), AP_large fluctuates due to fewer large-object samples.

4. References

[1] E. Hoffer, T. Ben-Nun, I. Hubara, N. Giladi, T. Hoefler, and D. Soudry, “Augment your batch: Improving generalization through instance repetition,” in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2020, pp. 8126–8135.

[2] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, and B. Guo, “Swin transformer: Hierarchical vision transformer using shifted windows,” in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, 2021, pp. 10012–10022.

[3] COCO API, “cocoeval.py,” GitHub, [Online]. Available: https://github.com/cocodataset/cocoapi/blob/master/PythonAPI/pycocotools/cocoeval.py

[4] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, “Feature pyramid networks for object detection,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2017, pp. 936–944.

[5] TorchVision, “_utils.py,” GitHub, [Online]. Available: https://github.com/pytorch/vision/blob/main/torchvision/models/_utils.py

[6] K. He, G. Gkioxari, P. Dollár, and R. Girshick, “Mask R-CNN,” in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, 2017, pp. 2961–2969.

[7] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards real-time object detection with region proposal networks,” in *Advances in Neural Inf. Process. Syst. (NeurIPS)*, vol. 28, 2015.

[8] A. Buslaev, V. I. Iglovikov, E. Khvedchenya, A. Parinov, M. Druzhinin, and A. A. Kalinin, “Albumentations: Fast and flexible image augmentations,” *Information*, vol. 11, no. 2, p. 125, 2020.

[9] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia, “Path aggregation network for instance segmentation,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2018, pp. 8759–8768.

[10] N. Moshkov, Y. Shvets, V. Davydenko, A. Kalinin, and V. Iglovikov, “Test-time augmentation for deep learning-based cell segmentation on microscopy images,” *Scientific Reports*, vol. 10, no. 1, p. 5068, 2020.

5. Additional experiments

I choose Mask R-CNN with Swin Transformer as the backbone, FPN + PAN as the neck, and the default Mask R-CNN head. This configuration balances feature richness and multi-scale fusion, and is especially suitable for instance segmentation of small or densely packed objects.

Pros:

1. Swin Transformer Backbone for Fine-Grained Features

   Compared to traditional CNNs like ResNet-50, Swin introduces window-based attention and hierarchical structure, allowing it to better capture small-scale and fine-grained object details.

2. FPN + PAN for Enhanced Multi-Scale Feature Fusion

   - FPN passes semantic features top-down to refine object detail at different scales.
   - PAN complements this by-passing spatial detail bottom-up, improving the early-stage features missed by FPN.

Cons:

1. Training Efficiency and Resource Usage

   Swin Transformer models are generally heavier in terms of GPU memory usage and computational cost compared to CNN backbones. This results in longer training times and higher hardware requirements.

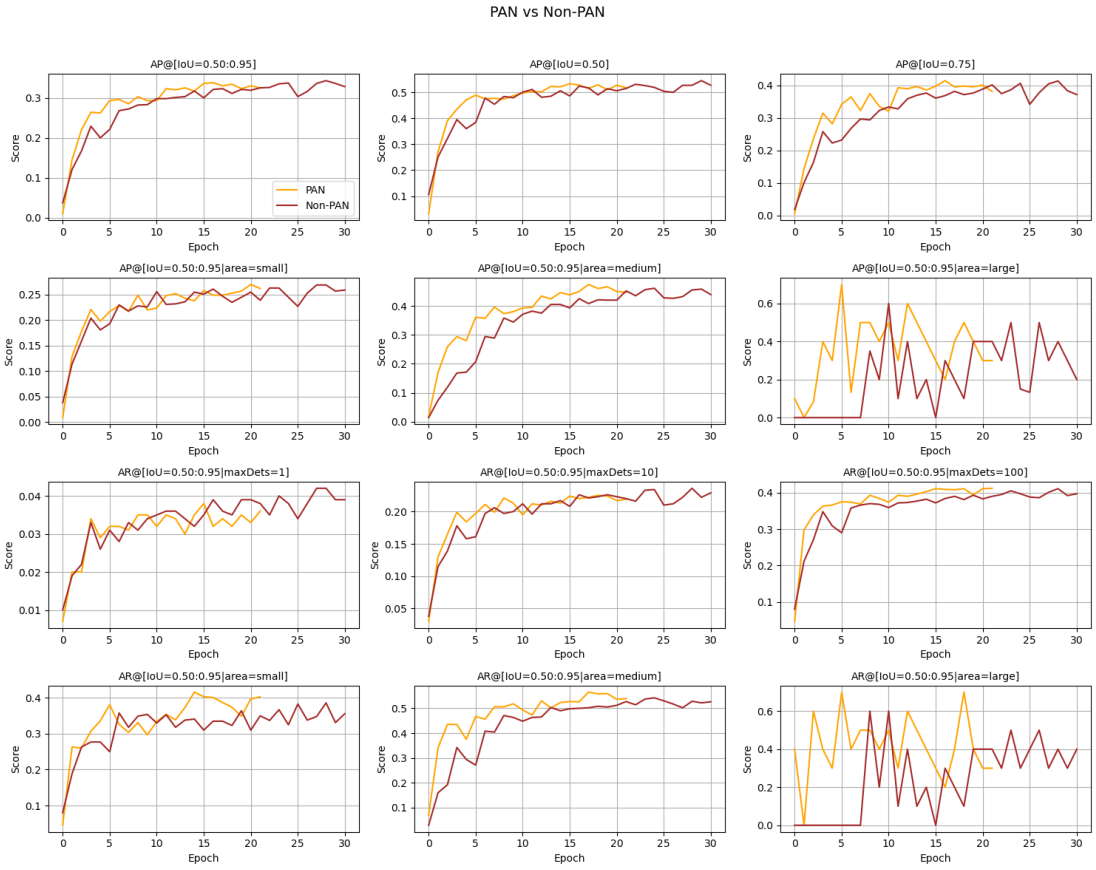
2. Longer Convergence Time

   The integration of both FPN and PAN with a transformer backbone may increase the complexity of feature fusion, making the network slower to converge without careful tuning.

Hypothesis 1: We hypothesize that even with a transformer-based backbone like Swin V2, which already captures long-range dependencies, adding a Path Aggregation Network (PAN) on top of the FPN can further enhance multi-scale feature fusion and localization cues, particularly improving detection performance on small objects.

How this may work: Swin V2 leverages window-based self-attention to effectively model both local and global context, while FPN constructs a top-down feature pyramid to propagate semantic information. PAN adds a complementary bottom-up path, reinforcing spatial detail propagation that FPN lacks. This may refine Swin V2's instance segmentation ability on fine-grained objects. However, since transformers already have strong integration mechanisms, the benefit of PAN may vary depending on the dataset and task complexity.

Results and Implications:



As shown in the results, integrating PAN into the Swin V2 backbone consistently leads to superior performance on key metrics such as AP@[0.50:0.95], AP_small, and AR_small. These improvements suggest that PAN effectively complements Swin V2's self-attention-based representation by reinforcing fine-grained spatial details that are critical for detecting small and densely packed instances.

Moreover, the PAN-enhanced model exhibits faster convergence within the first 10 epochs, highlighting its ability to stabilize and enrich multi-scale feature fusion at early stages of training. This accelerated learning curve indicates that PAN not only boosts final performance but also enhances training efficiency through improved feature flow across scales.

As shown in the results, integrating PAN into the Swin V2 backbone consistently leads to superior performance on key metrics such as AP@[0.50:0.95], AP_small, and AR_small. These improvements suggest that PAN effectively complements Swin V2's self-attention-based representation by reinforcing fine-grained spatial details that are critical for detecting small and densely packed instances.

Moreover, the PAN-enhanced model exhibits faster convergence within the first 10 epochs, highlighting its ability to stabilize and enrich multi-scale feature fusion at

early stages of training. This accelerated learning curve indicates that PAN not only boosts final performance but also enhances training efficiency through improved feature flow across scales.