[Github link]

Introduction

Task Overview

Instance segmentation is a challenging computer vision task that requires not only classifying objects within an image but also delineating the precise pixel boundaries for each distinct object instance. This homework focuses on applying instance segmentation to colored medical images, specifically to identify and segment four types of cells. The dataset consists of 209 images for training/validation and 101 images for testing, provided in .tif format.

My core approach utilizes the Mask R-CNN framework [1], a powerful and widely adopted model for instance segmentation. I leverage a ResNet-50 [4] (or ResNet-50 V2) backbone with a Feature Pyramid Network (FPN) [5] neck. To potentially enhance the segmentation accuracy, particularly for cluttered or touching instances, I introduce auxiliary prediction heads for cell centers and boundaries, drawing inspiration from techniques used in specialized biomedical image segmentation tools like CellPose [3]. The hypothesis is that explicitly learning these features will guide the model to produce more accurate and well-separated instance masks.

Method Overview

The model is based on **Mask R-CNN** using a **ResNet-50 or ResNet-50 V2** backbone with a **Feature Pyramid Network (FPN)** for multi-scale feature extraction. Standard heads for box and mask prediction were retained, with custom predictors to support 4 cell classes.

To improve segmentation accuracy, especially in cluttered regions, two optional auxiliary heads were added: a **center map head** and a **boundary map head**. These heads, inspired by CellPose, aim to provide spatial cues for better mask separation.

The total loss combines standard Mask R-CNN losses with optional auxiliary losses (weighted BCE loss for center and boundary maps). Training used the AdamW optimizer, cosine annealing scheduler, and elastic deformation as the main augmentation.

Data Preprocessing

Data Augmentation

The raw image and mask data are provided in .tif format. Each unique pixel value in the ground truth masks represents a distinct instance within each class-specific .tif file. () The preprocessing pipeline involves loading images and masks, generating auxiliary target maps (center and boundary), and applying data augmentations.

Loading Images and Instance Masks

- RGB images are loaded from image.tif for each sample in the dataset.
- For each of the four classes (class1.tif to class4.tif), if the mask file exists, it's read using skimage.io.imread.
- Within each class mask, individual instances (identified by unique pixel values greater than 0) are processed to create binary masks.
- Bounding boxes (boxes) are derived from these binary instance masks. Each instance is assigned its corresponding class label (labels).
- This process is handled within the __getitem__ method of the CellDataset class in utils_torchvision.py. If no objects are present in an image, empty tensors are created for masks, boxes, and labels to ensure model compatibility.

Center and Boundary Map Generation

- Auxiliary target maps, center_map and boundary_map, are pre-generated using the script get_train_map.py and saved as .npy files in the ./data/train_maps directory. These maps are then loaded by the CellDataset.
- Merged Instance Mask (load_and_merge_masks): Before generating center/boundary maps, masks from all four classes for a given image are merged into a single mask where each unique pixel value represents a unique instance across all classes. The scipy.ndimage.label function is used to ensure distinct instance IDs even if different class masks originally used overlapping ID numbers.
- **Center Heatmap Generation** (generate_center_heatmap):
    - For each unique instance in the merged mask, its center of mass is calculated using scipy.ndimage.center_of_mass.
    - A heatmap is initialized to zeros. At the calculated integer coordinates of the center of mass for each instance, a value of 1.0 is set.
    - This point-wise heatmap is then smoothed using a Gaussian filter (scipy.ndimage.gaussian_filter) with sigma=5.
    - Finally, the heatmap is normalized by its maximum value to ensure pixel values are between 0 and 1. This process is inspired by techniques used in keypoint detection [6].

- **Boundary Map Generation** (generate_boundary_map):
    - For each unique instance in the merged mask:
        - A binary mask for the current instance is created.
        - The boundary is found by taking the XOR (^) operation between the dilated version (scipy.ndimage.binary_dilation) and the eroded version (scipy.ndimage.binary_erosion) of the instance mask. This creates a 1-pixel thick boundary.
    - The boundary maps for all instances are combined using a maximum operation to form the final boundary map for the image.
- These pre-generated maps (shape (1, H, W)) are loaded and passed along with the image and standard instance masks to the augmentation pipeline.

Data Augmentation

- I utilized the albumentations library [7] for data augmentation during training. The validation set uses normalization and tensor conversion only (get_val_transform).
- The following augmentations were applied sequentially to the training images, their corresponding instance masks, and the generated center_map and boundary_map (specified via additional_targets in A.Compose):
    - A.ElasticTransform(alpha=30, sigma=12, p=0.3): This transformation introduces local pixel shifts, mimicking non-rigid deformations often seen in biological tissues. The alpha parameter controls the displacement intensity, and sigma controls the smoothness of the displacement field. It is applied with a probability of 0.3.
    - A.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]): Standard ImageNet normalization constants are used to normalize the pixel values of the input image. ()
    - ToTensorV2(): Converts the augmented image and masks/maps into PyTorch tensors, with the image being transposed from HWC to CHW format.

Model Architecture & Training Strategy

Model Summary

My model is based on Mask R-CNN [1] implemented in torchvision [8].

- **Backbone:**
    - I used a ResNet-50 [4] (or ResNet-50 V2, specify which one you used, e.g., resnet50_v2) pretrained on ImageNet [2].
    - The model_type argument in model.py allows selection between resnet50 and resnet50_v2.
- **Neck:**
    - A Feature Pyramid Network (FPN) [5] is used as the neck. FPNs are effective at detecting objects at different scales by combining features from multiple levels of the backbone.
- **Heads:**
    - **Region Proposal Network (RPN):** Part of the standard Mask R-CNN, responsible for proposing candidate object regions.
    - **Box Predictor Head:**
        - The original box predictor head from the pretrained Mask R-CNN is replaced with a FastRCNNPredictor.
        - The input features are model.roi_heads.box_predictor.cls_score.in_features.
        - The output is adapted to num_classes (which is 5: background + 4 cell types).
    - **Mask Predictor Head:**
        - The original mask predictor head is replaced with a MaskRCNNPredictor.
        - The input channels are model.roi_heads.mask_predictor.conv5_mask.in_channels.
        - The hidden dimension is 256 (default for Mask R-CNN), and the output is adapted to num_classes.
    - **Auxiliary Heads (Center and Boundary):**
        - When with_train_map is true, two additional heads (center_head and boundary_head) are added. These are instances of the ExtraHead class defined in model.py.
        - The ExtraHead consists of two 2D convolutional layers: the first maps in_channels (from the mask features, specifically model.roi_heads.mask_predictor.conv5_mask.in_channels) to 256 channels with a ReLU activation, and the second maps 256 channels to 1 output channel (binary prediction for center/boundary).

- These heads operate on features extracted by the backbone (specifically, the first feature map from the backbone's output dictionary, as seen in train.py).
- The motivation for these heads is inspired by approaches like CellPose [3], which demonstrate that predicting related topological or geometric features can aid instance segmentation.

Training Setup

Optimizer	`optim.AdamW(filter(lambda p: p.requires_grad, model.parameters()), lr=0.0001, weight_decay=1e-4)`
Scheduler	`optim.lr_scheduler.CosineAnnealingLR( optimizer, T_max = num_epoochs )`

Loss	The total loss is a combination of standard Mask R-CNN losses and losses from the auxiliary heads.
	• Mask R-CNN Losses: These include classification loss, bounding box regression loss, and mask prediction loss, as defined by He et al. [1]. These are automatically calculated by the torchvision Mask R-CNN model.
	• Auxiliary Head Losses:
	  ○ For the `center_head` and `boundary_head`, a Binary Cross-Entropy with Logits loss (`nn.BCEWithLogitsLoss`) is used.
	  ○ The target center and boundary maps are resized to match the prediction dimensions using bilinear interpolation for center maps and nearest neighbor interpolation for boundary maps (`F.interpolate`).
	  ○ The individual center and boundary losses are weighted by `w_center` and `w_boundary` respectively (default 0.5 each) and added to the main Mask R-CNN loss.
	  ○ The formula for the combined loss is: $L_{MaskRCNN} + w_{center} \cdot L_{center} + w_{boundary} \cdot L_{boundary}$

LR	**1e-4**
Batch size	Tried on 1, 2, 4. No big difference.

Evaluation

I use TorchMetrics' MeanAveragePrecision to compute:

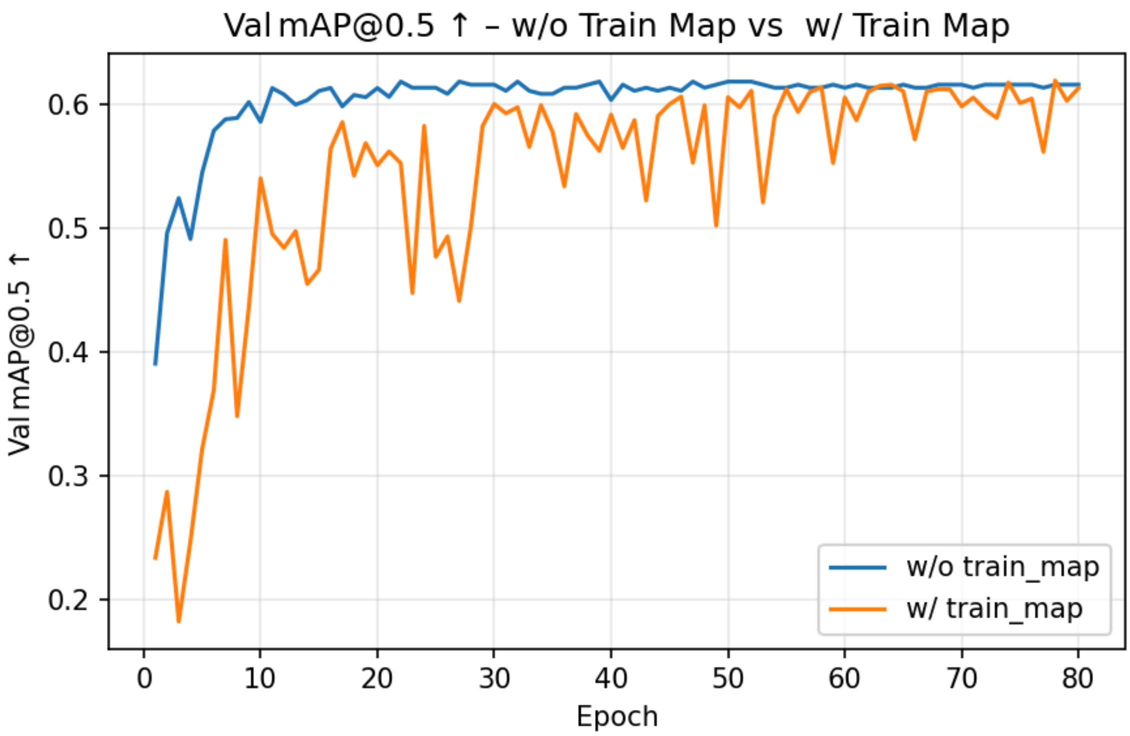
- **mAP@[.5:.95]**: averaged over multiple IoU thresholds from 0.5 to 0.95.
- **mAP@0.5**: IoU threshold fixed at 0.5.

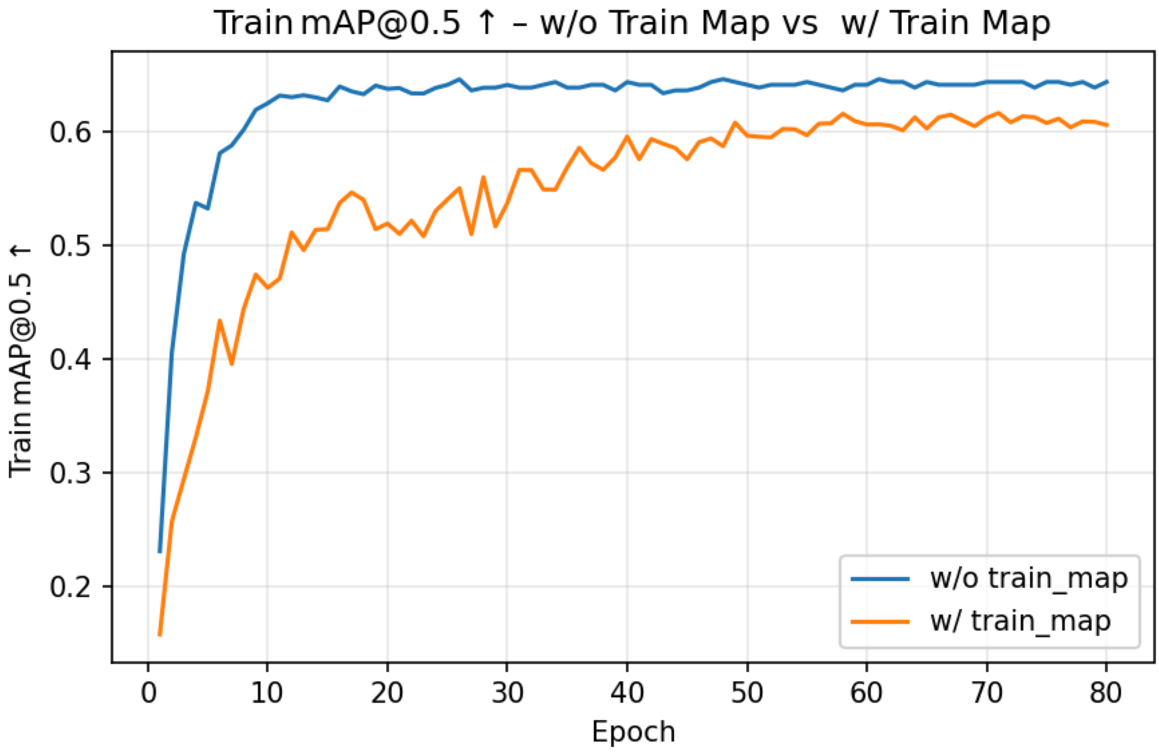
I recorded both values for training and validating phases.

Experiment

Initially, I applied rotation and flipping for data augmentation. While the training loss and validation mAP appeared reasonable, the results on the leaderboard were unexpectedly poor. As a result, I switched to using only elastic transformation, which is commonly employed in medical image segmentation tasks. However, the performance didn't go better with elastic transform.

The following plots show the training log comparison between ResNet50_v2 backbone trained with or without center map and boundary map. I set the weight of both auxiliary maps as 0.5.





According to the graph, using train_map hurts both training and validation performance, especially in stability and early convergence. Although final validation performance reaches similar levels, the path to convergence is noisier with train_map. Without train_map, training is smoother, faster, and generalization to validation is more reliable. I'm still experimenting with other weight combinations, which might yield better results.

Conclusion

Adding center and boundary supervision increased training instability and slowed convergence without improving final performance. Models **without auxiliary heads** trained more smoothly and generalized better. Further tuning of auxiliary loss weights is ongoing and may improve results in future experiments.

References

[1] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in IEEE International Conference on Computer Vision (ICCV), 2017. (If using arXiv: arXiv:1703.06870)
[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2009.
[3] C. Stringer, T. Wang, M. Michaelos, and M. Pachitariu, "Cellpose: a generalist algorithm for cellular segmentation," Nature Methods, vol. 18, no. 1, pp. 100–106, 2021. (or arXiv:2006.09913)

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.
[5] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature Pyramid Networks for Object Detection," in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017.
[6] X. Zhou, D. Wang, and P. Krähenbühl, "Objects as Points," arXiv:1904.07850, 2019.
[7] 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. (or GitHub: https://github.com/albumentations-team/albumentations)
[8] TorchVision Team, "TorchVision: PyTorch's Computer Vision Companion," GitHub: https://github.com/pytorch/vision, [Year of version used].
[9] I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization," in International Conference on Learning Representations (ICLR), 2019. (or arXiv:1711.05101)
[10] I. Loshchilov and F. Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts," in International Conference on Learning Representations (ICLR), 2017. (or arXiv:1608.03983) (Note: CosineAnnealingLR is often attributed to this paper or presented as a variant of cyclical learning rates.)
[11] A. Paszke, et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in Advances in Neural Information Processing Systems 32 (NeurIPS), 2019.