Homework 3: Visual Recognition using Deep Learning - Instance Segmentation

Sophie FU (313554802)

1 Introduction

This project addresses the task of instance segmentation on a medical image dataset composed of four distinct cell types. The objective is to identify and segment individual cell instances from input `.tif` images using a vision-based deep learning model.

A Mask R-CNN architecture was selected due to its strong performance in instance segmentation. The model was trained on 209 annotated training images and evaluated on a test set of 101 unlabelled images. Predictions were formatted in the COCO standard, which includes bounding boxes, category identifiers, confidence scores, and segmentation masks represented in Run-Length Encoding (RLE).

Final results were submitted to the Codabench evaluation platform, where model performance was assessed using mean Average Precision (mAP). Additional experiments were conducted to investigate potential improvements through architectural changes and training modifications.

The full source code and output files are available on GitHub at:
`https://github.com/Soophiie/DeepLearning-Homework-3`

2 Method

2.1 Dataset and Preprocessing

The dataset consists of 209 training samples and 101 test images, each provided in `.tif` format. For each training image, up to four binary mask files (`class1.tif` to `class4.tif`) are available, with each file containing instance-level annotations for a specific cell class. Each unique pixel value within a mask corresponds to a separate instance of that class.

The dataset is structured with one folder per image, containing the raw image and the corresponding class masks. During preprocessing, the masks were loaded using `skimage.io.imread`, and individual binary masks were extracted for each instance and class. These masks were then encoded using Run-Length Encoding (RLE) with the `pycocotools` library to match the COCO segmentation format.

The RGB images were converted to float tensors and normalized to the $[0, 1]$ range. No resizing or padding was applied, as the model was designed to accept inputs of varying dimensions. Since no separate validation set was provided, a portion (15%) of the training data was randomly reserved for validation using `random_split`.

For the test set, the image filenames were mapped to image IDs using the provided `test_image_name_to_ids.json`. Only predictions with a confidence score above 0.3 were retained for submission. The final output was saved in a single `test-results.json` file, containing `image_id`, `category_id`, `score`, and `segmentation` fields.

2.2 Model Architecture

The base model used for instance segmentation is the Mask R-CNN implementation from the `torchvision` library, adapted from the official PyTorch repository [1]. Mask R-CNN was chosen due to its strong performance in instance segmentation tasks and its ability to simultaneously predict bounding boxes and pixel-level masks. This is particularly valuable in biomedical imaging, where objects can overlap or present complex shapes requiring precise delineation.

The architecture consists of a backbone network, a feature pyramid network (FPN), and a set of region-of-interest (ROI) heads. A ResNet-50 backbone pretrained on ImageNet was selected for feature extraction and integrated with an FPN to improve multi-scale object detection.

To adapt the model to the dataset, the classification and bounding box heads were replaced using `FastRCNNPredictor`, and the mask prediction head was replaced using `MaskRCNNPredictor`. These modifications allow the model to output predictions for five classes (four cell types and background).

To explore the impact of backbone depth, an alternative version of the model was also implemented with a ResNet-101 backbone. While this deeper architecture provides increased representational power, it also requires more computation and training time. This idea was inspired by the cyclic refinement approach proposed in [2], which motivates the use of deeper networks to iteratively improve instance-level segmentation quality.

The final model outputs, for each image, include bounding boxes, class probabilities, and binary masks. Predictions were filtered based on confidence scores before being encoded in the COCO-style submission format.

One known limitation of Mask R-CNN is its computational overhead and suboptimal performance on small or overlapping objects. These challenges are especially relevant in the context of medical imaging, where cellular structures may be densely packed or partially occluded.

2.3 Training Settings

The model was trained using the PyTorch framework on Google Colab with GPU acceleration enabled. The training process was carried out for 10 to 15 epochs, depending on the experiment.

A batch size of 2 was used due to GPU memory limitations. The optimizer selected was Stochastic Gradient Descent (SGD) with a learning rate of 0.005, momentum of 0.9, and

weight decay of 0.0005. Learning rate scheduling was handled by `CosineAnnealingLR` or `ReduceLROnPlateau`, depending on the experiment.

The loss function used was the default multi-task loss of Mask R-CNN, which combines classification loss, bounding box regression loss, and mask segmentation loss.

Since no validation set was provided, 15% of the training data was randomly split and reserved for validation. Validation loss was monitored at each epoch to select the best model, which was saved whenever the validation performance improved.

To support generalization, a small set of geometric transformations (random horizontal and vertical flips) was applied in one of the experiments through a custom `SimpleTransformTorch` class. As described in Section 2.1, all images were normalized to the range $[0, 1]$ before being fed to the network.

To better interpret model behavior and performance, visualizations were generated every five epochs on a small batch of randomly selected samples. For each image, three visual representations were provided: the ground truth mask, the predicted mask overlaid on the input image with confidence scores and bounding boxes, and the combined prediction mask. These visual plots facilitated qualitative assessment of segmentation quality and failure cases throughout training.

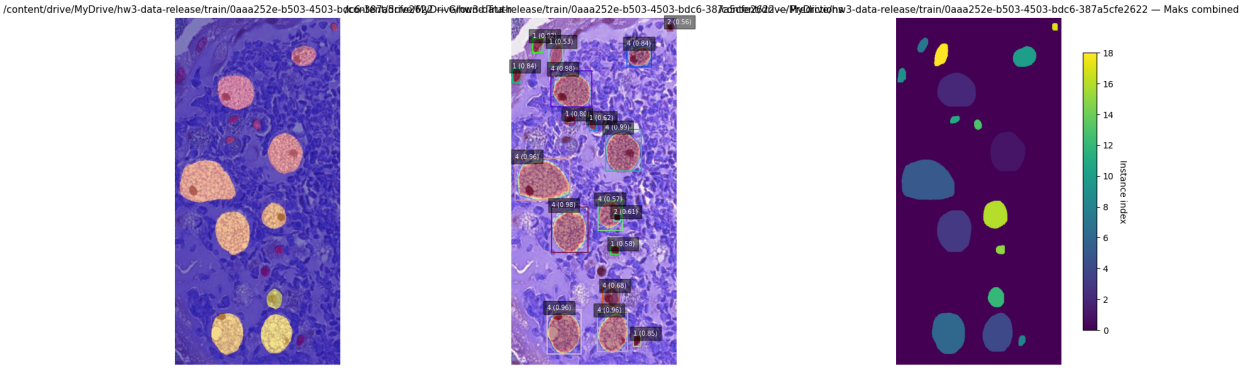


Figure 1: Example visualization generated during training. From left to right: input image with ground truth mask overlay, predicted mask with bounding boxes and confidence scores, and combined predicted mask with instance indices.

Model checkpoints and performance metrics such as training and validation loss were logged, and loss curves were plotted to monitor convergence and overfitting trends.

A minimal set of data augmentation techniques was implemented using a custom transformation class, inspired by examples from the official Torchvision documentation [3].

3 Experiments and Results

3.1 Metrics and Evaluation

The primary evaluation metric used for this task is the mean Average Precision (mAP) at an Intersection over Union (IoU) threshold of 0.5, commonly denoted as AP@0.5. This metric measures both the localization and segmentation accuracy of the predicted instances relative to ground truth annotations.

Predictions were evaluated on the Codabench platform. Each prediction included the following elements in COCO format: `image_id`, `category_id`, `bbox` (bounding box in XYWH format), `score`, and `segmentation` (in Run-Length Encoding format).

Before submission, predictions were filtered to retain only instances with confidence scores above 0.3. This threshold was chosen empirically to reduce false positives while preserving relevant predictions.

The final submission file, `test-results.json`, was validated locally and then uploaded to Codabench. The output distribution across predicted categories was visualized to verify class coverage and balance. Qualitative comparisons between predicted and ground truth masks were also generated to assess segmentation quality visually.

In addition to mAP, training and validation loss curves were analyzed to monitor convergence and potential overfitting across epochs.

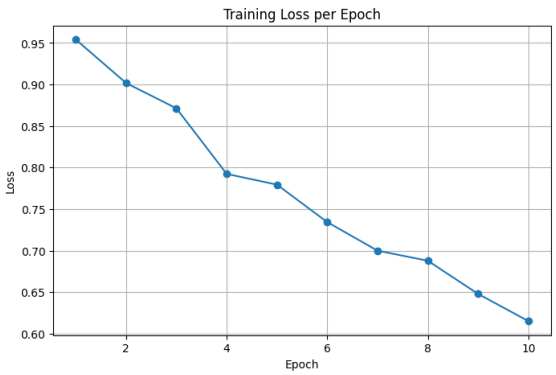
3.2 Additional Experiments

To explore potential improvements in segmentation performance, several experimental variants of the baseline model were implemented.

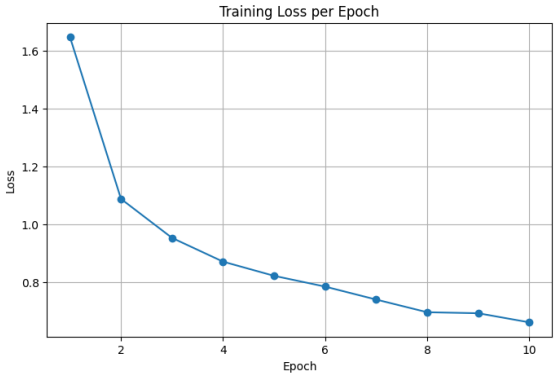
1. **Optimizer and Scheduler Tuning:** The optimizer was changed from SGD to AdamW, and various learning rate schedulers such as `ReduceLROnPlateau` and `CosineAnnealingLR` were tested. This configuration led to the most consistent improvement in validation loss and training stability, although the overall gain remained modest.

2. **Backbone Modification:** The ResNet-50 backbone was replaced with ResNet-101 to enhance feature representation. However, this change did not lead to improved accuracy or mAP scores. Training time increased considerably, and the larger model appeared more prone to overfitting on the small dataset.

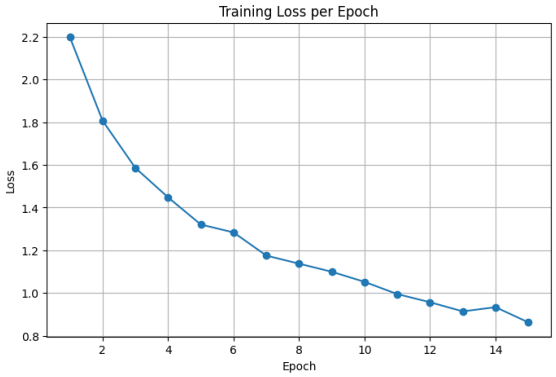
3. **Data Augmentation:** Random horizontal and vertical flips were introduced as basic data augmentation. Although this helped stabilize training slightly, it did not yield measurable improvements in segmentation performance.



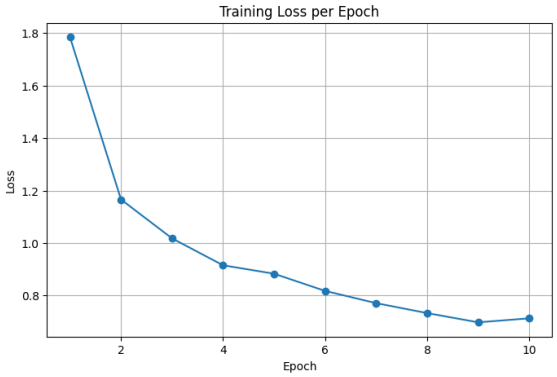
(a) Original loss curve



(b) Training loss curve for experiment 1



(c) Training loss curve for experiment 2



(d) Training loss curve for experiment 3

Figure 2: Comparison of training loss curves across baseline and three experimental configurations.

Overall, the only configuration that produced a marginal improvement over the baseline was the optimizer and scheduler adjustment (experiment 1). Other architectural modifications did not outperform the original setup. This result reinforces a common observation in instance segmentation tasks: simple and well-regularized models, such as the original Mask R-CNN with a ResNet-50 backbone, often achieve the best balance between performance and generalization—especially when working with limited or domain-specific data.

4 Submission Details

The final predictions were generated using the best-performing model, selected based on the lowest validation loss. During inference, each test image was preprocessed and passed through the model to obtain bounding boxes, masks, class scores, and predicted labels.

Only predictions with a confidence score above 0.3 were retained to reduce false positives. Binary masks were thresholded at 0.5 and encoded using Run-Length Encoding (RLE) via the `encode_mask` function provided in the `utils.py` module.

The output for each instance included `image_id`, `category_id`, `bbox`, `score`, and `segmentation`. The final results were stored in a single file named `test-results.json`, following

the COCO format as required by the Codabench evaluation platform.

The model was evaluated on Codabench using the private leaderboard. Before submission, the format and content of the JSON file were manually verified, and class distributions were analyzed to ensure balanced predictions across categories. The entire codebase and output file were uploaded to the GitHub repository, and a link to the repository is included in this report.

To ensure the submission file contained balanced and meaningful predictions, the distribution of predicted categories in the `test-results.json` file was analyzed. As shown in Figure 3, each predicted class is represented, and the number of instances varies depending on the complexity and frequency of cell types in the test set. This qualitative check helped confirm that no category was entirely missing from the model's predictions, and provided additional insights into class-level model behavior.

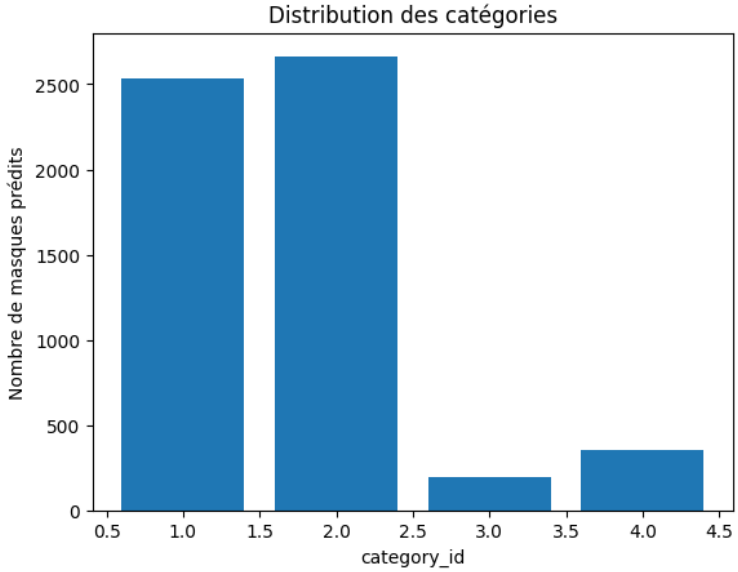


Figure 3: Distribution of predicted classes in the `test-results.json` file. Each bar indicates the number of instances predicted for a given category.

References

[1] PyTorch, *Torchvision mask r-cnn implementation*, `https://github.com/pytorch/vision/blob/main/torchvision/models/detection/mask_rcnn.py`, Accessed: 2025-05-07, 2024.

[2] Z. Zhao, Q. Cao, J. C. Niebles, and S. Savarese, "Cyclic cell instance segmentation with object-centric refinement," *arXiv preprint arXiv:2012.07177*, 2020.

[3] T. Contributors, *End-to-end object detection with transforms — pytorch tutorials*, `https://docs.pytorch.org/vision/main/auto_examples/transforms/plot_transforms_e2e.html`, Accessed: 2025-05-07, 2024.