HW3 Instance Segmentation

王嘉羽
313551052

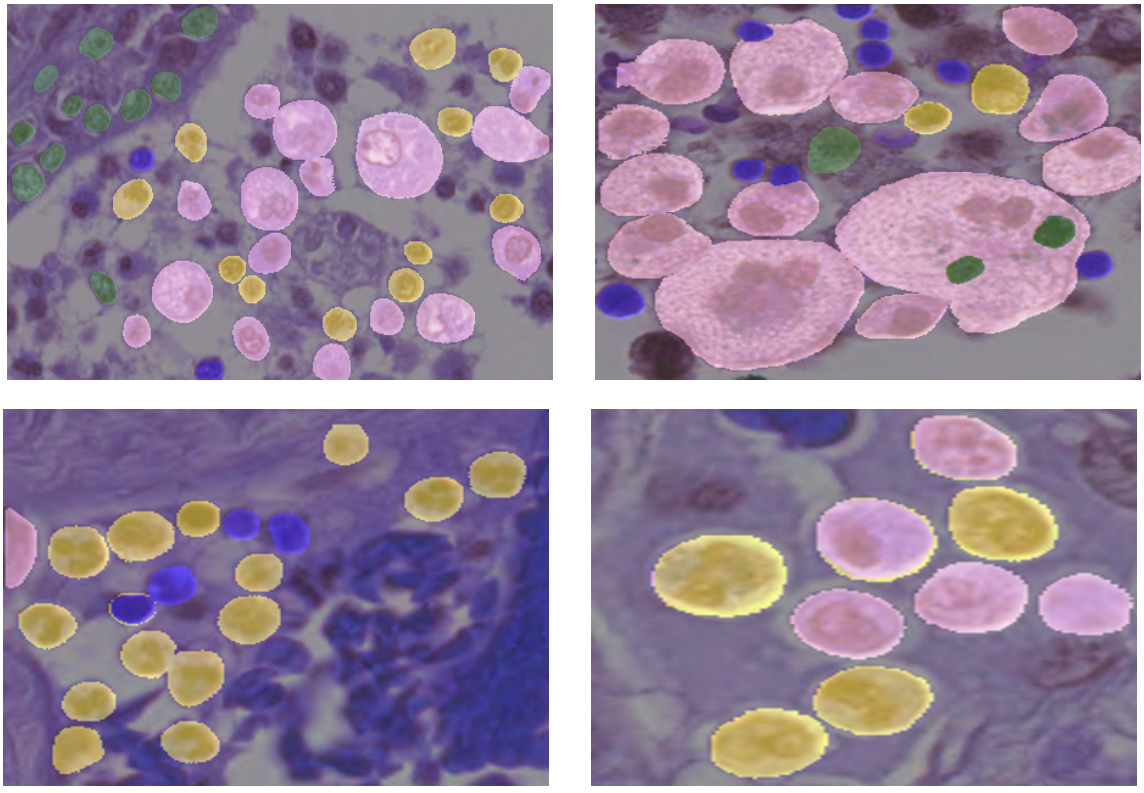
1 Introduction

In this assignment, we aim to train a deep learning model for cell instance segmentation using a biomedical image dataset provided for the HW3 Cell Segmentation task. The objective is to accurately detect and segment up to four types of cells in each image, though some images may contain fewer types. The final segmentation results are encoded using run-length encoding (RLE) and follow the COCO dataset format.

I employed the Mask R-CNN [1] model with a ResNet-50 FPN v2 [2] backbone. During experimentation, I explored various model adjustments, such as modifying the FPN structure, tuning anchor sizes, and adjusting the learning rate schedule through different schedulers. For image preprocessing, I simply converted all images to grayscale to reduce color variance and simplify input channels.

The raw dataset was first converted to the COCO format with instance-level annotations. After training and tuning, the final model achieved a mean average precision (mAP) score of **0.3781** on the Public test, demonstrating reasonable segmentation performance on complex biomedical images.

Github: https://github.com/vayne1125/NYCU-Visual-Recognitionusing-Deep-Learning/tree/main/HW3_Instance-Segmentation



2 Method

The parameters I selected are based on the results of experiments, and detailed information can be found in ” 3 Experiments” and ” 4 Additional Experiments” .

2.1 Pre-processing

The original dataset was first converted to the COCO format and saved as a JSON file to enable compatibility with standard PyTorch training pipelines. This conversion facilitates efficient loading and parsing of annotations during training.

The following transformations were applied to the training images:

- ❖ $\mathrm{T.ToTensor}()$
  Converts the image into a PyTorch tensor and scales pixel values to the range [0, 1].
- ❖ $\mathrm{T.Normalize}(\mathrm{mean}{=}[0.485,\ 0.456,\ 0.406],\ \mathrm{std}{=}[0.229,\ 0.224,\ 0.225])$
  Applies normalization using standard ImageNet statistics to stabilize training and improve convergence.

2.2 Model Architecture

In this project, I adopted the Mask R-CNN [1] architecture, which consists of three main components: a backbone, a neck (feature pyramid), and task-specific heads. To better suit the task of small object detection in cell images, I made several modifications to create a more lightweight and efficient variant of the model.

The model uses a ResNet50-FPN v2 [2] as the backbone. The anchor generator was customized with smaller anchor sizes to improve sensitivity to small instances. Additionally, the box RoI pooling layer was modified to use fewer feature map levels ([’0’, ’1’]) to reduce computational overhead, which also improved performance during validation. The number of output classes was adapted to the dataset, and both the box and mask heads were reconfigured accordingly.

The implementation is shown below (total parameters: 45896557):

```python
def light_maskrcnn_v2(num_classes):
    """
    Returns a lightweight Mask R-CNN V2 model customized for small object detection.
    """
    anchor_sizes = ((4,), (8,), (16,), (32,), (64,))
    aspect_ratios = ((0.5, 1.0, 2.0),) * len(anchor_sizes)
    model = torchvision.models.detection.maskrcnn_resnet50_fpn_v2(
        weights=MaskRCNN_ResNet50_FPN_V2_Weights.DEFAULT,
        min_size=800,
        max_size=1333,
```

```
        rpn_anchor_sizes=anchor_sizes,
        rpn_aspect_ratios=aspect_ratios
    )
    model.roi_heads.box_roi_pool = MultiScaleRoIAlign(
        featmap_names=['0', '1'],
        output_size=7,
        sampling_ratio=2
    )
    in_features_box = model.roi_heads.box_predictor.cls_score.in_features
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features_box, 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
    )
    return model
```

Listing 1: Implementation of the `light_maskrcnn_v2`

2.3 Hyperparameter Settings

The following hyperparameters were used in training:

Hyperparameter	Value
Optimizer	SGD
Learning Rate	0.005
Momentum	0.9
Weight Decay	0.0005
Scheduler	WarmupCosineLR (Listing 2)
Total Iters	len(train_loader) * num_epochs
Warmup Iters	total_iters // 20
Model	light_maskrcnn_v2(num_classes=5) (Listing 1)
Epochs	50
Batch Size	4
Early Stop	10

Table 1: Hyperparameter Settings

```python
class WarmupCosineLR(torch.optim.lr_scheduler._LRScheduler):
    def __init__(self, optimizer, warmup_iters, total_iters, last_epoch=-1):
        self.warmup_iters = warmup_iters
        self.total_iters = total_iters
        super(WarmupCosineLR, self).__init__(optimizer, last_epoch)

    def get_lr(self):
        if self.last_epoch < self.warmup_iters:
            # Warmup phase: Linear increase
            return [
                base_lr * float(self.last_epoch + 1) / self.warmup_iters
                for base_lr in self.base_lrs
            ]
        else:
            # Cosine Annealing phase
            cos_iter = self.last_epoch - self.warmup_iters
            cos_total = self.total_iters - self.warmup_iters
            return [
                base_lr * 0.5 * (1 + math.cos(math.pi * cos_iter / cos_total))
                for base_lr in self.base_lrs
            ]
```

Listing 2: Implementation of the `WarmupCosineLR` Learning Rate Scheduler

This custom learning rate scheduler combines a **linear warm-up phase** followed by a **cosine annealing decay**.

- During the initial `warmup_iters`, the learning rate increases linearly from 0 to the base learning rate.
- After warm-up, it decays following a cosine function until `total_iters`.

This design helps stabilize training in the early stages (preventing sudden large updates) and gradually reduces the learning rate to encourage convergence. It's a commonly used strategy in training deep models, especially when the initial learning rate is relatively large.

3 Experiments

In this section, we describe the experiments conducted to optimize the model by adjusting key hyperparameters, with a focus on comparing different optimizers and learning rate schedulers.

Two different optimizer-scheduler combinations were explored to determine the most effective training strategy. The first setup used **Stochastic Gradient Descent (** `SGD` **)** [3] with a `WarmupCosineLR` [4] scheduler (see Listing 2). The optimizer was configured with a learning rate of 0.005, momentum of 0.9, and weight decay of 0.0005. The learning rate schedule followed a **warm-up phase** for the first 5% of the total iterations, followed by a **cosine annealing phase**. This strategy was implemented as follows:

```
total_iters = len(train_loader) * num_epochs
warmup_iters = total_iters // 20
optimizer = torch.optim.SGD(
    model.parameters(),
    lr=0.005,
    momentum=0.9,
    weight_decay=0.0005
)
scheduler = WarmupCosineLR(optimizer, warmup_iters=warmup_iters, total_iters=total_iters)
```

The second setup utilized the `AdamW` optimizer, which has shown strong performance in HW2 Digit Recognition Problem. It was configured with a learning rate of 0.0001 and weight decay of 0.0001, with the scheduler set to `StepLR`. This approach multiplies the learning rate by a factor of 0.1 after every 3 epochs:

```
LEARNING_RATE = 0.0001
WEIGHT_DECAY = 0.0001
optimizer = optim.AdamW(model.parameters(), lr=LEARNING_RATE, weight_decay=WEIGHT_DECAY)

GAMMA = 0.1
STEP_SIZE = 3
scheduler = StepLR(optimizer, step_size=STEP_SIZE, gamma=GAMMA)
```

4 Additional Experiments

This section investigates the effect of modifying input data and architectural components of the Mask R-CNN model, specifically the neck and head modules.

4.1 Data Processing

In this project, which focuses on cell instance segmentation, I hypothesized that color information might be less informative for biomedical cell images compared to structural and morphological features. In many cell segmentation tasks, the key discriminative cues lie in the shapes, boundaries, and textures of the cells rather than their color. Therefore, I conducted grayscale preprocessing experiments, expecting the model to focus more effectively on morphological patterns by removing potentially noisy or irrelevant color channels.

To implement this, I modified the data transformation pipeline as follows:
From:

```
train_transform = T.Compose([
    T.ToTensor(),
    T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

To:

```
transform = T.Compose([
    T.Grayscale(num_output_channels=1),  # Convert to grayscale
    T.ToTensor(),                        # Convert to tensor with shape 1xHxW
    T.Normalize(mean=[0.5], std=[0.5])   # Normalize with single-channel statistics
])
```

4.2 Mask R-CNN Architecture

This section describes the modifications made to the Mask R-CNN architecture, focusing on adjustments to the neck and head components to better suit the characteristics of the cell instance segmentation task.

❖ **Neck Adjustments**
Anchor Size Comparison: Three anchor size configurations were tested to improve detection performance for small cell instances:

- **DEFAULT**: $\text{anchor\_sizes} = ((8,), (16,), (32,), (64,), (128,))$
- **SMALL ONE**: $\text{anchor\_sizes} = ((4,), (8,), (16,), (32,), (64,))$
- **EVEN SMALLER**: $\text{anchor\_sizes} = ((2,), (4,), (8,), (16,), (32,), (64,))$

These variations aim to provide finer anchor granularity, which is particularly beneficial for detecting small and densely packed objects such as individual cells.

❖ **Head Adjustments**
RoI Pooling Feature Map Selection: To reduce noise and enhance localization on high-resolution feature maps, I adjusted the number of feature map levels used in the Region of Interest (RoI) pooling layer.

Original configuration:

```
model.roi_heads.box_roi_pool = torchvision.ops.MultiScaleRoIAlign(
    featmap_names=['0', '1', '2', '3'],
    output_size=7,
    sampling_ratio=2
)
```

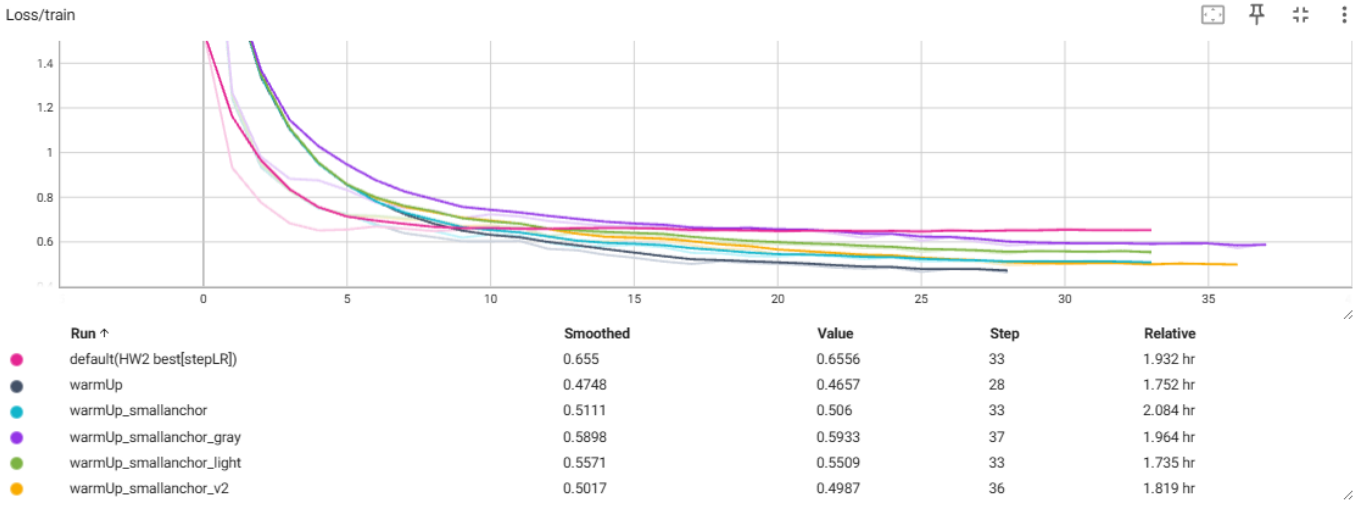
Modified configuration:

```
model.roi_heads.box_roi_pool = torchvision.ops.MultiScaleRoIAlign(
    featmap_names=['0', '1'],
    output_size=7,
    sampling_ratio=2
)
```

This change was motivated by the nature of the dataset, where most cell boundaries and features are already well-preserved in high-resolution feature maps. By focusing only on earlier layers (e.g., '0' and '1'), the model avoids incorporating coarse, low-resolution features from deeper layers that may dilute detail and reduce precision in segmentation.

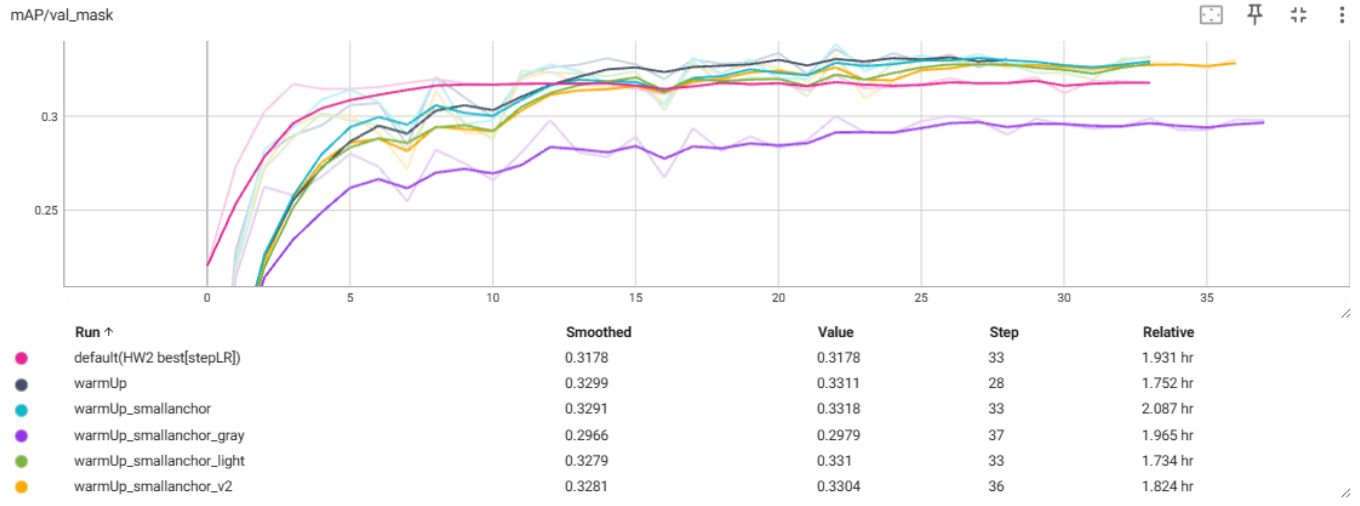
5 Results

The performance of all experiments is evaluated using three primary metrics: training loss, validation mAP, and public mAP. The final model for submission is selected based on the highest public mAP score. A visual comparison of training loss and validation mAP across different experiments is shown in Figure 1. The detailed descriptions of each experiment, along with their corresponding validation mAP and public mAP, are summarized in Table 3.



Run	Smoothed	Value	Step	Relative
default(HW2 best[stepLR])	0.655	0.6556	33	1.932 hr
warmUp	0.4748	0.4657	28	1.752 hr
warmUp_smallanchor	0.5111	0.506	33	2.084 hr
warmUp_smallanchor_gray	0.5898	0.5933	37	1.964 hr
warmUp_smallanchor_light	0.5571	0.5509	33	1.735 hr
warmUp_smallanchor_v2	0.5017	0.4987	36	1.819 hr

(a) Training Loss across experiments.



Run	Smoothed	Value	Step	Relative
default(HW2 best[stepLR])	0.3178	0.3178	33	1.931 hr
warmUp	0.3299	0.3311	28	1.752 hr
warmUp_smallanchor	0.3291	0.3318	33	2.087 hr
warmUp_smallanchor_gray	0.2966	0.2979	37	1.965 hr
warmUp_smallanchor_light	0.3279	0.331	33	1.734 hr
warmUp_smallanchor_v2	0.3281	0.3304	36	1.824 hr

(b) Validation mAP across experiments.

Figure 1: Overall comparison of training loss and validation mAP across different model variations and settings.

Experiment Name	Validation mAP	Public mAP
default(HW2 best[StepLR])	0.3215	0.3252
warmUp	0.3311	0.3748
warmUp_smallAnchor	0.3383	0.3778
warmUp_smallAnchor_v2	0.3325	0.3599
warmUp_smallAnchor_gray	0.3001	0.3537
warmUp_smallAnchor_light	0.3314	**0.3781**

Table 2: Summary of experiment configurations and their corresponding Validation and Public mAP scores.

5.1 Comparison of Different Optimizers and Learning Rate Schedulers

At the beginning, I assumed this assignment would be quite similar to HW2, so I started with the same learning rate schedule: `StepLR` with the same set of parameters. As a second approach, I used `SGD` [3] with a `WarmupCosineLR` scheduler [4] —a strategy I found to be the most commonly adopted in segmentation tasks based on online research and assistance from GPT.

Before running the experiments, I hypothesized that the standard `SGD` [3] strategy would yield better results, as it is a widely used and stable baseline. The outcomes confirmed this assumption. The performance comparison between the two strategies can be seen in the default and warmUp rows of Table 3.

5.2 Comparison with and without Grayscale Data Preprocessing

In this experiment, the results contradicted my initial expectations. I originally assumed that reducing color information through grayscale preprocessing would help the model better focus on boundaries, which are crucial in segmentation tasks. However, the results showed that this approach performed the worst in both validation mAP and public mAP.

Additionally, from the training loss shown in Figure 1, we can observe that the grayscale model converged the slowest. This suggests that, for this task, color information is essential. Removing color cues prevents the model from accurately identifying the positions of cells.

The impact of grayscale preprocessing can be observed by comparing the warmUp_smallanchor and warmUp_smallanchor_gray rows in Table 3.

5.3 Comparison of Mask R-CNN Architectures (Anchor Sizes)

In HW2, reducing the anchor sizes significantly improved accuracy due to the small object nature of digits. Following the same strategy in this task, I experimented with three different anchor size configurations:

- **DEFAULT**: anchor_sizes = ((8,), (16,), (32,), (64,), (128,))
  (Corresponds to warmUp)
- **SMALL ONE**: anchor_sizes = ((4,), (8,), (16,), (32,), (64,))
  (Corresponds to warmUp_smallanchor)
- **EVEN SMALLER**: anchor_sizes = ((2,), (4,), (8,), (16,), (32,), (64,))
  (Corresponds to warmUp_smallanchor_v2)

Before running the experiments, I assumed that SMALL ONE and EVEN SMALLER would perform similarly, as I believed the lower bound of effective anchor sizes had already been reached.

The results confirmed this hypothesis. SMALL ONE outperformed EVEN SMALLER, suggesting that adding an extremely small anchor size (i.e., 2) may have led to over-segmentation of larger cells, thereby harming performance.

The full comparison can be found in Figure 1 and Table 3, particularly in the rows labeled warmUp, warmUp_smallanchor, and warmUp_smallanchor_v2.

5.4 Comparison of Mask R-CNN Architecture (ROI Head)

This experiment builds upon a similar approach from HW2, where the number of layers in the roi_heads was reduced. While the purpose in HW2 was to improve accuracy, the motivation here is different: due to limited computational resources, I aimed to investigate whether reducing the number of layers could achieve comparable accuracy in less training time.

Surprisingly, the simplified version not only reduced training time by approximately 0.35 hours, but it also achieved a higher public mAP, although the validation mAP was slightly lower. A summary of training time and accuracy metrics is shown in Table 3.

I believe this improvement in public mAP can be attributed to the reduced capacity of the ROI head, which led to less overfitting on the validation set. With fewer layers, the model may have generalized better to the unseen public test set. On the other hand, the slight drop in validation mAP suggests that the simplified model may have had slightly less capacity to capture complex patterns in the validation set.

Experiment Name	Time	Validation mAP	Public mAP
warmUp_smallAnchor	2.087 hr	**0.3383**	0.3778
warmUp_smallAnchor_light	**1.734 hr**	0.3314	**0.3781**

Table 3: Comparison of training time, validation mAP, and public mAP between standard and lightweight ROI head architectures.

Based on the results of the aforementioned experiments, the configuration warmUp_smallAnchor_light was selected as the final submission model due to its superior balance of accuracy and training efficiency. The visualized predictions of the selected model are presented in Figure 2.

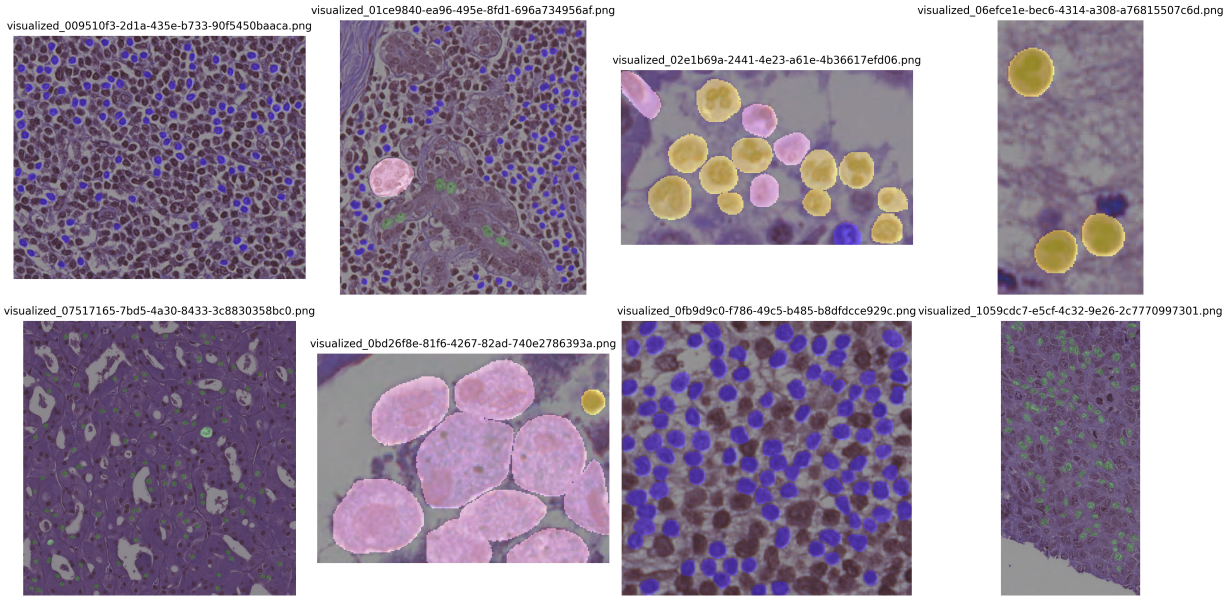


Figure 2: Qualitative results of the final model (warmUp_smallAnchor_light) on test images.

References

[1] K. He, G. Gkioxari, P. Dollár, and R. Girshick, “Mask r-cnn,” 2018. [Online]. Available: https://arxiv.org/abs/1703.06870

[2] Y. Li, S. Xie, X. Chen, P. Dollar, K. He, and R. Girshick, “Benchmarking detection transfer learning with vision transformers,” 2021. [Online]. Available: https://arxiv.org/abs/2111.11429

[3] S. Ruder, “An overview of gradient descent optimization algorithms,” 2017. [Online]. Available: https://arxiv.org/abs/1609.04747

[4] I. Loshchilov and F. Hutter, “Sgdr: Stochastic gradient descent with warm restarts,” 2017. [Online]. Available: https://arxiv.org/abs/1608.03983