HW2 Digit Recognition Problem

王嘉羽
313551052

1 Introduction

In this assignment, we aim to train a deep learning model for digit recognition using a dataset provided for the HW2 Digit Recognition Problem. Our goal is to accurately detect the location of each digit in an image and recognize which digit it is.

To achieve this, I experimented with the Faster R-CNN model using a ResNet-50 FPN backbone. I modified the FPN structure, including tuning the anchor sizes to better match the scale of the digits in the dataset. Additionally, I explored various data preprocessing strategies, such as binarization and data augmentation, to improve model robustness and generalization.

The dataset consists of training, validation, and test sets, formatted in COCO style with multiple bounding boxes and digit class annotations per image. The model is trained using a multi-task objective, optimizing both localization and classification accuracy.

Our final model achieves a task1 score of **0.36** and a task2 score of **0.77**, both surpassing the strong baseline benchmarks. These results demonstrate the model's effectiveness in handling both digit detection and recognition under the given dataset conditions.

Github: https://github.com/vayne1125/NYCU-Visual-Recognitionusing-Deep-Learning/tree/main/HW2_Digit-Recognition-Problem

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-process data

I experimented with color-based data augmentation and image binarization, but neither performed as well as using the original images. In the end, applying only normalization yielded the best results.

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

2.2 Model architecture

In this assignment, we adopt the Faster R-CNN architecture [1], which consists of three main components: the **Backbone**, the **Neck (RPN)**, and the **Head (ROI Heads)**.

- **Backbone:** We use a pretrained ResNet-50 model combined with a Feature Pyramid Network (FPN). ResNet-50 extracts high-level semantic features, while the FPN enhances multi-scale feature representation, which is particularly beneficial for detecting small objects like digits.
- **Neck (RPN):** A customized anchor generator is used with anchor sizes set to `((4,), (8,), (12,), (24,), (48,))` and aspect ratios of ((0.25, 0.5, 1.0),) to better match the typical height-width ratios of digits. The RPN is responsible for generating object proposals for further classification and regression.
- **Head (ROI Heads):**
  - MultiScaleRoIAlign is applied to unify the feature maps from different FPN levels into a fixed-size representation.
  - The configuration includes hyperparameters such as `positive_fraction = 0.25` , `batch_size_per_image = 512` , `box_nms_thresh = 0.3` , and `box_score_thresh = 0.5` , among others. A detailed overview of the full architectural setup is provided below.
  - Finally, a custom `FastRCNNPredictor` is used with the number of output classes set to 11 (10 digits + 1 background class).

  Total Parameters: 41345286

```
anchor_sizes = ((4,), (8,), (12,), (24,), (48,))
aspect_ratios = ((0.25, 0.5, 1.0),) * len(anchor_sizes)

anchor_generator = AnchorGenerator(sizes=anchor_sizes, aspect_ratios=aspect_ratios)
```

```
model = torchvision.models.detection.fasterrcnn_resnet50_fpn(
    weights=FasterRCNN_ResNet50_FPN_Weights.DEFAULT,
    min_size=512,  # Default: 600
    max_size=1024, # Default: 1000
    rpn_anchor_generator=anchor_generator
)

model.roi_heads.positive_fraction = 0.25  # Default 0.5
model.roi_heads.batch_size_per_image = 512  # Default 512
model.roi_heads.box_nms_thresh = 0.3  # Default 0.5
model.roi_heads.box_score_thresh = 0.5  # Default 0.05
model.roi_heads.box_roi_pool = torchvision.ops.MultiScaleRoIAlign(
    featmap_names=['0', '1', '2', '3'],
    output_size=7,
    sampling_ratio=2
)
```

Listing 1: Architecture configuration code

2.3 Hyperparameter Settings

The following hyperparameters were used in training:

Hyperparameter	**Value**
Optimizer	AdamW
Learning Rate	0.0001
Weight Decay	0.0001
Scheduler	StepLR
Step Size	3
Gamma	0.1
Model	ftrcnn(num_classes=NUM_CLASSES)
Epochs	20
Batch Size	8
Early Stop	5

Table 1: Hyperparameter Settings

3 Experiments

This section outlines the hyperparameter tuning experiments conducted to optimize the model. A total of two experiments were performed, each focusing on adjusting a single hyperparameter. The results of these experiments are presented in Section 5, ”Results.”

- Comparison between `StepLR(3, 0.1)` and `CosineAnnealingLR` with a learning rate of 1×10^{-7} and $T_{\text{max}} = 20$.
- Comparison between batch sizes of 4 and 8.

4 Additional Experiments

This section focuses on experiments aimed at adjusting the input data and modifying the architecture of the Faster R-CNN model, specifically the neck and head components.

4.1 Data Processing

I experimented with data augmentation by altering the color and performing binarization beforehand.

- **Data Augmentation:**
  - T.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1)
  - T.RandomGrayscale(p=0.1)

  I believe that data augmentation can improve the model’s learning capacity, as it showed significant improvements in HW1. Therefore, I decided to include this experiment.

- **Adaptive Binarization:**

```
thresh = cv2.adaptiveThreshold(img, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY_INV,
    11, 2)
```

  I believe that binarizing the image beforehand can reduce the distractions caused by color variations, allowing the model to focus more on the ’shapes’

4.2 Faster R-CNN Architecture

In this part, I focused on modifying the Faster R-CNN architecture, specifically its neck and head components.

- **Neck adjustments:**

- **Anchor size comparison:** $((4,), (8,), (16,), (32,), (64,))$ vs $((4,), (8,), (12,), (24,), (48,))$.
  My intuition was that using smaller anchor sizes could improve the model's ability to detect small objects such as digits.
- **Aspect ratio comparison:** $((1, 1.5, 2.0),)$ vs $((0.25, 0.5, 1.0),)$.
  Since the digits in the images are mostly vertically elongated, I hypothesized that vertical aspect ratios like $(0.25, 0.5, 1.0)$ would be more suitable and potentially improve detection accuracy.

❖ **Head adjustments:**

- **NMS threshold comparison:** $\text{box\_nms\_thresh} = 0.3$ vs 0.5.
  This experiment was exploratory in nature, intended to observe how a stricter NMS might affect detection results.
- **Score threshold comparison** $\text{box\_score\_thresh} = 0.05$ vs 0.5.
  Similar to the NMS experiment, this was based on curiosity to explore the trade-off between detecting more objects and reducing false positives.
- **RoI pooling feature map comparison:** I modified the box_roi_pool to use fewer feature map levels:
  From:

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

  To:

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

  My reasoning was that, since the digits in the images are relatively small, using deeper feature maps may introduce excessive background noise. Therefore, I wanted to see if using only the earlier, higher-resolution feature maps would help the model focus more effectively on the digit regions.

5 Results

In the results section, I conducted five sets of comparisons using different hyperparameter settings. For each configuration, I recorded the following metrics:

- Final training loss

- Validation mean Average Precision (mAP)
- Validation digit classification accuracy(a prediction is considered correct only if the digit is detected and the class is correct; predictions on non-existent digits or incorrect digit classes are counted as incorrect)
- Public leaderboard scores for Task 1 (mAP)
- Public leaderboard scores for Task 2 (accuracy)

5.1 Comparison of Hyperparameters

❖ **Comparison of `StepLR` and `CosineAnnealingLR` :** In my previous work on HW1, I used `CosineAnnealingLR` [2], as it was reported to provide more stable learning. `CosineAnnealingLR` gradually reduces the learning rate following a cosine schedule, which avoids abrupt changes and encourages smoother convergence, especially near local minima. Therefore, I initially expected `CosineAnnealingLR` to outperform `StepLR` , which reduces the learning rate at fixed intervals in a more abrupt fashion.

However, the results turned out differently. While `CosineAnnealingLR` led to slightly lower training loss, `StepLR` achieved better validation mAP, digit classification accuracy, and performance on the public test set.

By examining the training curves (see Figure 1), I suspect the reason might be that `StepLR` maintains a constant learning rate for a longer period at each stage, which may help the model converge more effectively within each learning rate phase. In contrast, the continuous decay in `CosineAnnealingLR` might reduce the learning rate too early, limiting the model's capacity to explore and optimize further in the later training stages. The comparison of validation mAP, public task 1 (mAP), and public task 2 (accuracy) is shown in Table 2.

	Validation mAP	**Public Task 1 (mAP)**	**Public Task 2 (Accuracy)**
CosineAnnealingLR	0.4535	0.35	0.76
StepLR	0.4588	0.36	0.77

Table 2: Comparison of Validation mAP, Public Task 1 (mAP), and Public Task 2 (Accuracy) for `CosineAnnealingLR` and `StepLR` .

❖ **Comparison between batch sizes 4 and 8:** I expected that a batch size of 8 would perform better, but the results showed that only the Validation mAP was better, and even then, the difference was small. The train loss and validation accuracy were better for batch size 4, and the public task results were similar for both. I believe that the smaller batch size (4) benefits from more iterations, which leads to faster loss and accuracy reduction. However, the overall performance (mAP) did not improve as much. Therefore, I conclude that, on balance, batch size 8 is the better choice. The results are shown in Table 3, and the training curves for loss, accuracy, and mAP are illustrated in Figure 2.

	Validation mAP	**Public Task 1 (mAP)**	**Public Task 2 (Accuracy)**
batch size = 4	0.4643	0.36	0.77
batch size = 8	0.4653	0.36	0.77

Table 3: Comparison of Validation mAP, Public Task 1 (mAP), and Public Task 2 (Accuracy) for batch size 4 and 8.

5.2 Comparison with and without Data Processing

I initially expected that data augmentation would improve performance, as it did in HW1. However, in HW2, the effect was actually detrimental. I suspect that modifying colors and grayscale may have misled the model. The model relies on color information, and by randomly altering colors, I might have weakened the relevant signals it uses for prediction.

Similarly, I thought binarization would help, but the results were not as expected. After reviewing the binarized images, I realized that the process did not clearly separate the digits from the background, which likely caused the model to be misled. The binarization results are shown in Figure 3. The comparison of the three approaches is summarized in Figure 4 and Table 4.

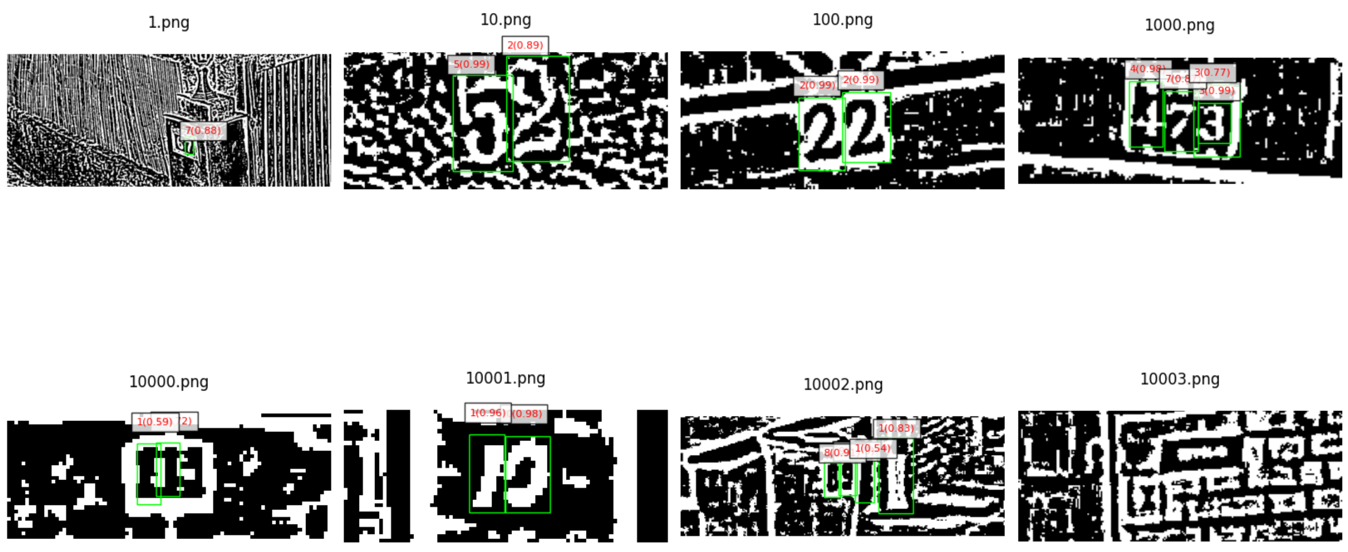


Figure 3: Binarization Prediction Results

5.3 Comparison of Faster R-CNN Architecture(Neck)

- **resnet50_fpn_stLR3_0.1_nms0.3_b4_e20** uses larger anchors (anchor_sizes = $((4,), (8,), (16,), (32,), (64,)))$.
- **resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_small_obj** uses smaller anchors (anchor_sizes = $((4,), (8,), (12,), (24,), (48,)))$.

	Validation mAP	Public Task 1 (mAP)	Public Task 2 (Accuracy)
None	0.4588	0.36	0.77
color	0.2545	-	-
binary	0.4309	0.32	0.63

Table 4: Comparison of Validation mAP, Public Task 1 (mAP), and Public Task 2 (Accuracy) for data processing.

- **resnet50_fpn_stLR3_0.1_nms0.3_b8_e20_small_obj_st0.5** uses horizontal anchors (aspect_ratios = ((0.25, 0.5, 1.0),)).
- **resnet50_fpn_stLR3_0.1_nms0.3_b8_e20_small_obj_st0.5_ver** uses vertical anchors (aspect_ratios = ((1.0, 1.5, 2.0),)).

I originally expected the smaller anchors and vertical aspect ratios to perform better. Indeed, the Validation mAP was better with the smaller anchors (since the digits are small) and vertical aspect ratios (as the digits are mostly vertically oriented). However, the Training Loss and Validation Accuracy did not improve as expected. I believe that the Validation mAP is a more accurate indicator of model performance. The results are shown in Figure 5 and Table 5.

	Validation mAP	Public Task 1 (mAP)	Public Task 2 (Accuracy)
Normal Anchor	0.4588	0.36	0.77
Small Anchor	0.4619	0.36	0.77
Horizonal Anchor	0.4653	0.36	0.77
Vertical Anchor	0.4671	0.36	0.77

Table 5: Comparison of Validation mAP, Public Task 1 (mAP), and Public Task 2 (Accuracy) for Different Neck Configurations.

5.4 Comparison of Faster R-CNN Architecture(Head)

❖ **Comparison of NMS and Score Threshold Settings:**

- **resnet50_fpn_stLR3_0.1_nms0.5_b4_e20** with NMS threshold = 0.5
- **resnet50_fpn_stLR3_0.1_nms0.3_b4_e20** with NMS threshold = 0.3
- **resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_small_obj** with score threshold = 0.05
- **resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_small_obj_st0.5** with score threshold = 0.5

Initially, I didn't have a specific strategy in mind, so I simply decided to adjust the parameters to see the effect. To my surprise, the results improved across the board. Upon further reflection, I realized that by setting a smaller NMS threshold, I allowed the model to predict more bounding boxes, which gave it more opportunities to detect objects. Meanwhile, by increasing the score threshold, I ensured that, in the absence of overly aggressive suppression of bounding boxes, the model could select the more accurate ones.

Interestingly, the model **resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_small_obj_st0.5** exhibited the slowest decrease in training loss. However, it performed exceptionally well in terms of validation mAP. The results, as shown in Figure 6 and Table 6, highlight this unexpected but promising outcome.

	Validation mAP	**Public Task 1 (mAP)**	**Public Task 2 (Accuracy)**
NMS threshold = 0.5	0.4611	0.36	0.76
NMS threshold = 0.3	0.4588	0.36	0.77
score threshold = 0.05	0.4619	0.36	0.77
score threshold = 0.5	0.4643	0.36	0.77

Table 6: Comparison of Validation mAP, Public Task 1 (mAP), and Public Task 2 (Accuracy) under Different NMS and Score Threshold Settings.

❖ **Comparison of RoI pooling feature map:** Regarding the RoI pooling feature map, I initially expected that using only the first two feature levels would yield comparable results to the default setting. Since the objects in this task are relatively small, they likely don't require as much contextual background, so it's reasonable to assume that the model primarily relies on the earlier layers. Reducing the number of levels should also improve computational efficiency.

However, based on the results, the model using only two levels performed worse overall. Although it showed better training performance, its validation and public performance declined. The comparison is shown in Figure 7 and Table 7.

	Validation mAP	**Public Task 1 (mAP)**	**Public Task 2 (Accuracy)**
4 layers	0.4653	0.36	0.77
2 layers	0.4633	0.35	0.76

Table 7: Comparison of Validation mAP, Public Task 1 (mAP), and Public Task 2 (Accuracy) under Different RoI Pooling Feature Map Settings.

Considering the findings from all the experiments above, I selected the following configuration as the final setup:

- No additional data processing applied

- Carefully chosen thresholds and anchor settings to balance detection sensitivity and accuracy
- A comprehensive RoI pooling strategy across multiple feature maps

The details of the final configuration are summarized in Table 8 and Listings 2:

Parameter	**Setting**
Data Processing	None
NMS Threshold	0.3
Score Threshold	0.5
Batch Size	8
Anchor Sizes	((4,), (8,), (12,), (24,), (48,))
Aspect Ratios	((0.25, 0.5, 1.0),)
RoI Pooling	MultiScaleRoIAlign

Table 8: Final configuration selected for submission.

The MultiScaleRoIAlign is configured as follows:

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

The final prediction results using the selected configuration are illustrated in Figure 8.

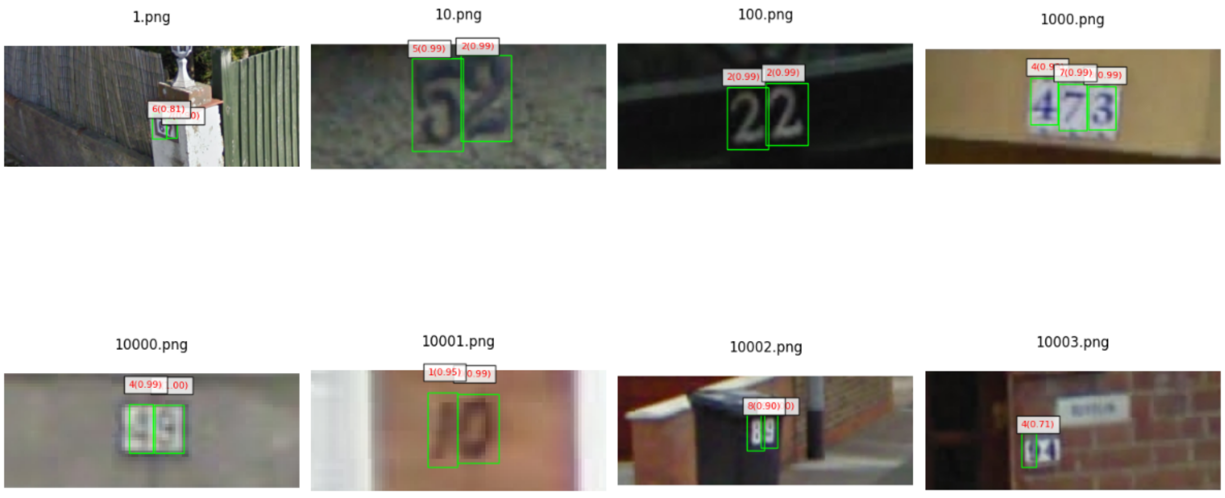
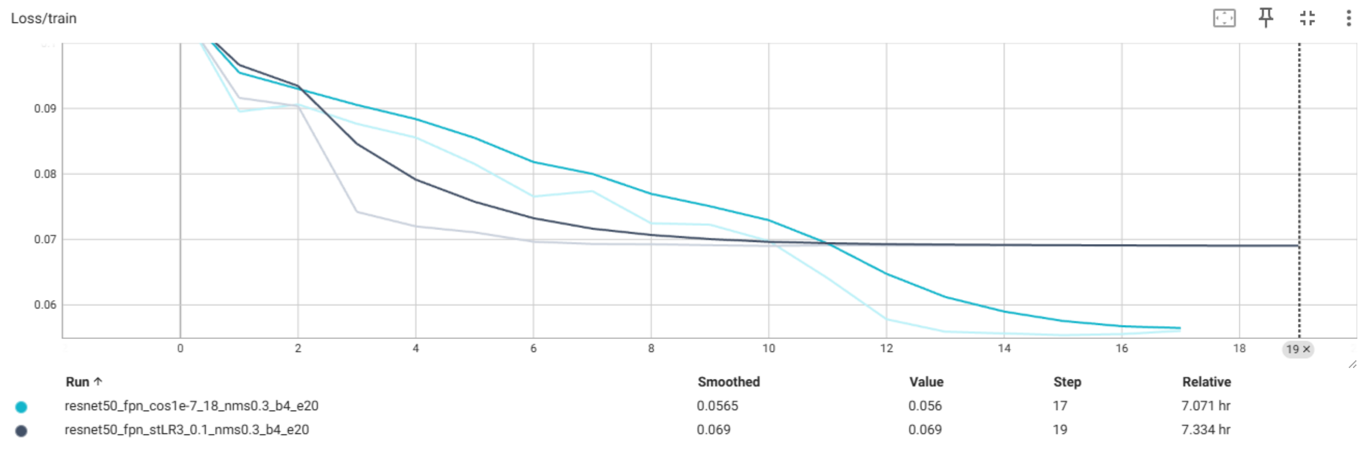
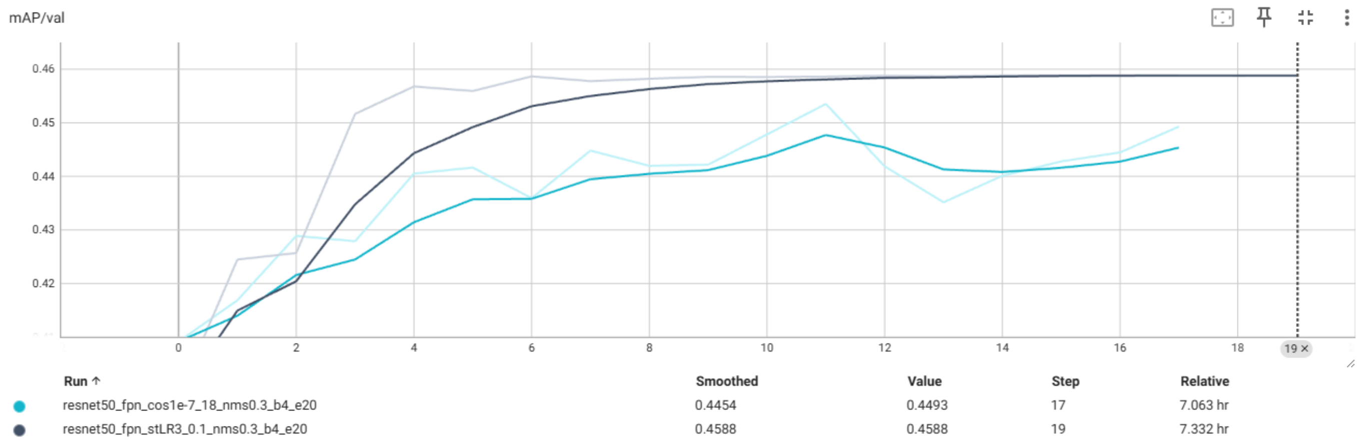


Figure 8: Final prediction results with the selected configuration.



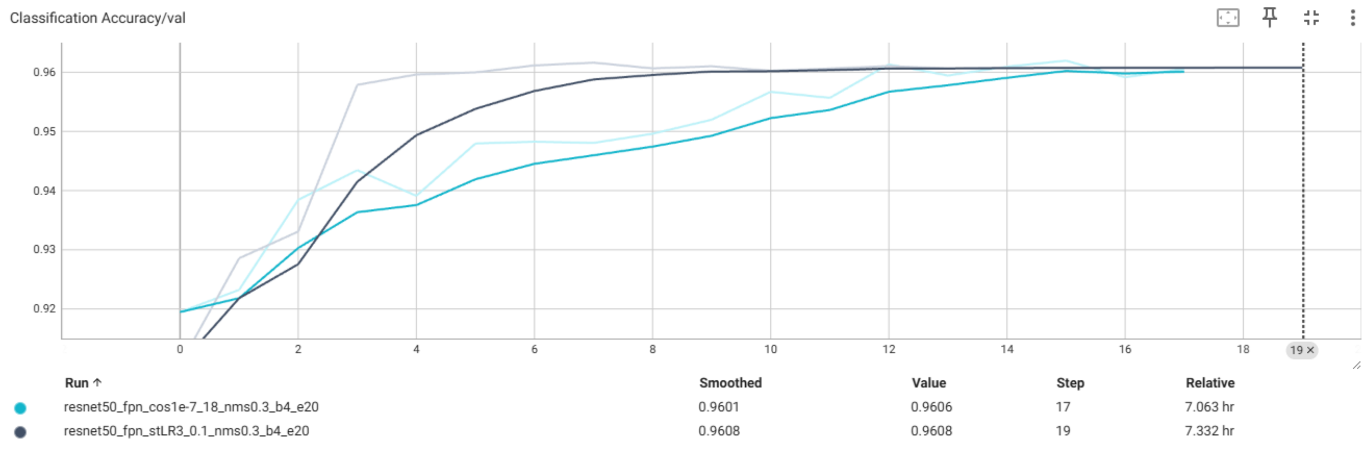
	Run ↑	Smoothed	Value	Step	Relative
●	resnet50_fpn_cos1e-7_18_nms0.3_b4_e20	0.0565	0.056	17	7.071 hr
●	resnet50_fpn_stLR3_0.1_nms0.3_b4_e20	0.069	0.069	19	7.334 hr

(a) Training Loss



	Run ↑	Smoothed	Value	Step	Relative
●	resnet50_fpn_cos1e-7_18_nms0.3_b4_e20	0.4454	0.4493	17	7.063 hr
●	resnet50_fpn_stLR3_0.1_nms0.3_b4_e20	0.4588	0.4588	19	7.332 hr

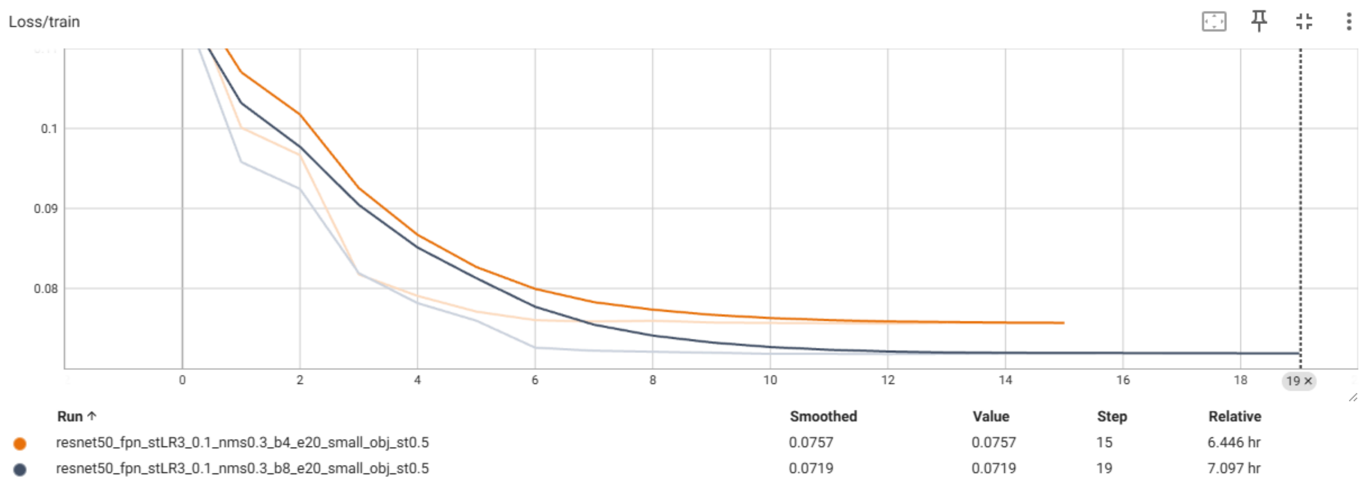
(b) Validation mAP



	Run ↑	Smoothed	Value	Step	Relative
●	resnet50_fpn_cos1e-7_18_nms0.3_b4_e20	0.9601	0.9606	17	7.063 hr
●	resnet50_fpn_stLR3_0.1_nms0.3_b4_e20	0.9608	0.9608	19	7.332 hr

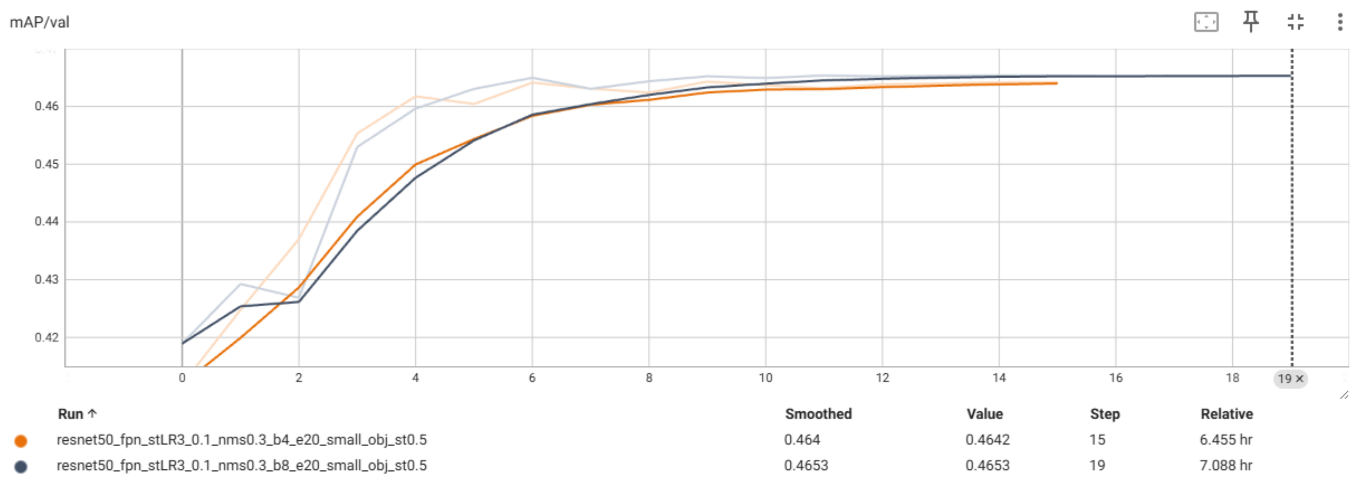
(c) Validation Accuracy

Figure 1: Overall performance comparison between `StepLR` and `CosineAnnealingLR` across training loss, mAP, and digit classification accuracy.



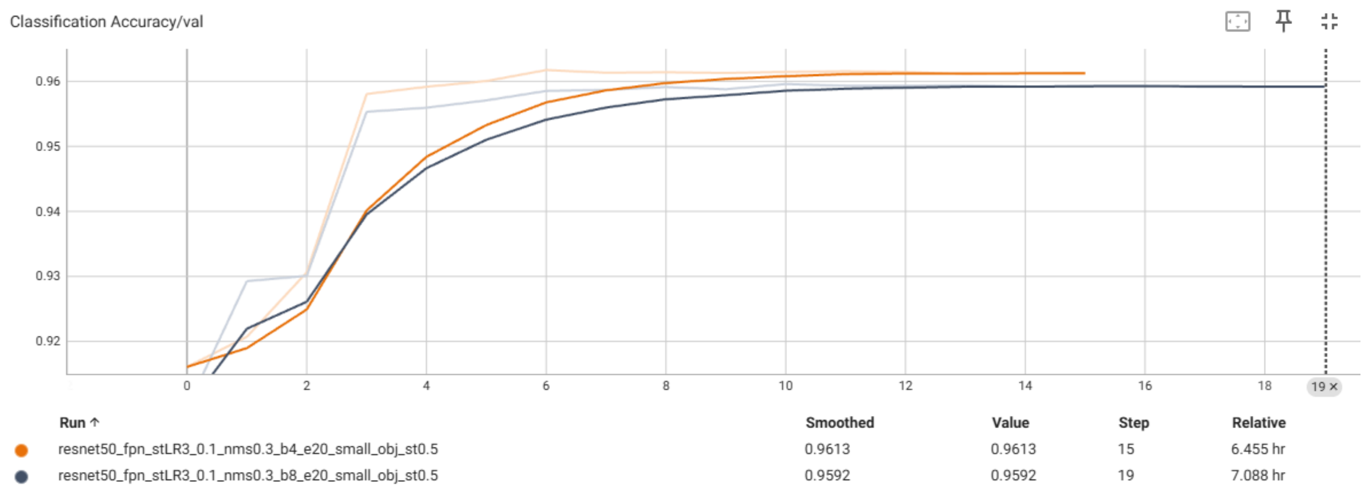
	Run ↑	Smoothed	Value	Step	Relative
●	resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_small_obj_st0.5	0.0757	0.0757	15	6.446 hr
●	resnet50_fpn_stLR3_0.1_nms0.3_b8_e20_small_obj_st0.5	0.0719	0.0719	19	7.097 hr

(a) Training Loss



	Run ↑	Smoothed	Value	Step	Relative
●	resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_small_obj_st0.5	0.464	0.4642	15	6.455 hr
●	resnet50_fpn_stLR3_0.1_nms0.3_b8_e20_small_obj_st0.5	0.4653	0.4653	19	7.088 hr

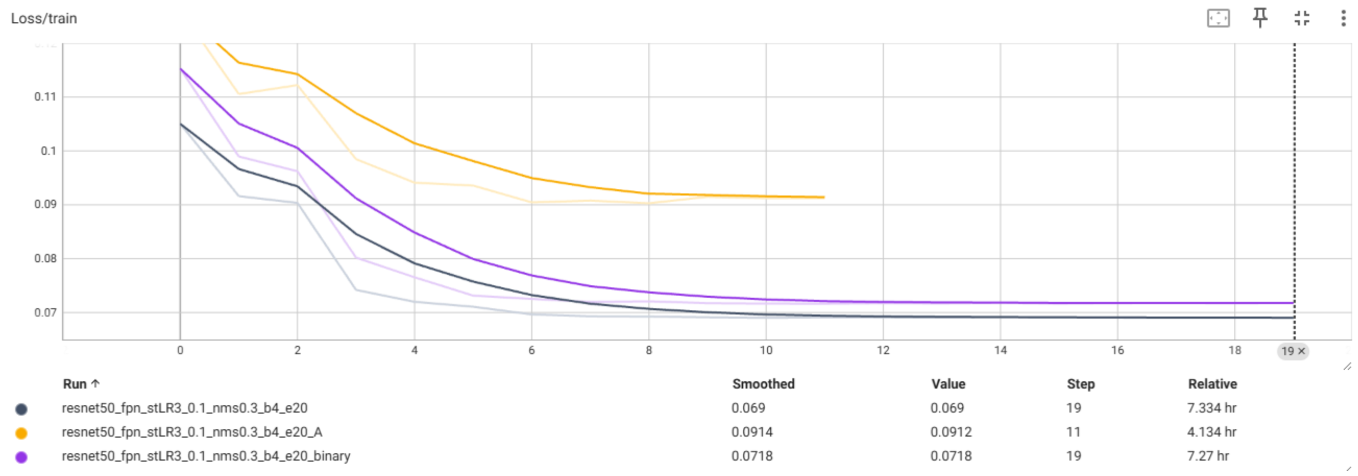
(b) Validation mAP



	Run ↑	Smoothed	Value	Step	Relative
●	resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_small_obj_st0.5	0.9613	0.9613	15	6.455 hr
●	resnet50_fpn_stLR3_0.1_nms0.3_b8_e20_small_obj_st0.5	0.9592	0.9592	19	7.088 hr

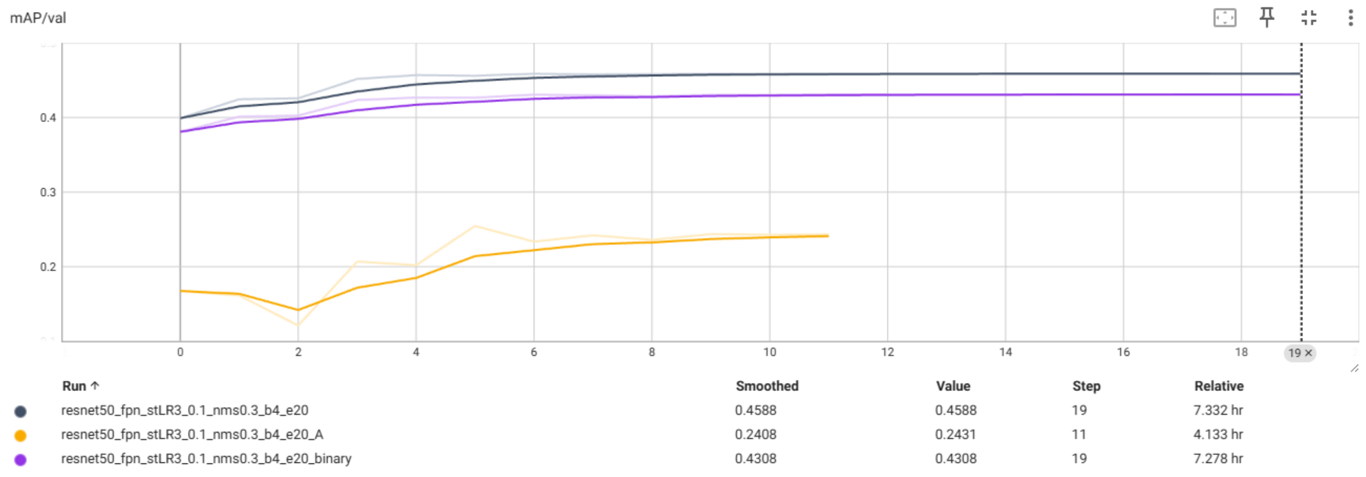
(c) Validation Accuracy

Figure 2: Overall performance comparison between batch size 4 and 8 across training loss, mAP, and digit classification accuracy.



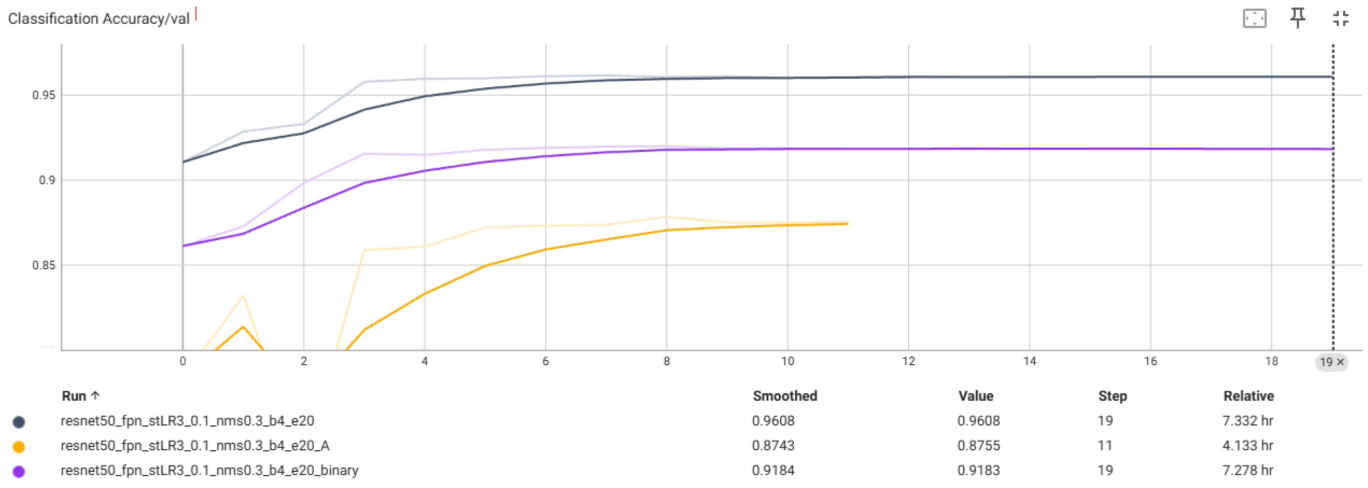
	Run ↑	Smoothed	Value	Step	Relative
●	resnet50_fpn_stLR3_0.1_nms0.3_b4_e20	0.069	0.069	19	7.334 hr
●	resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_A	0.0914	0.0912	11	4.134 hr
●	resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_binary	0.0718	0.0718	19	7.27 hr

(a) Training Loss



	Run ↑	Smoothed	Value	Step	Relative
●	resnet50_fpn_stLR3_0.1_nms0.3_b4_e20	0.4588	0.4588	19	7.332 hr
●	resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_A	0.2408	0.2431	11	4.133 hr
●	resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_binary	0.4308	0.4308	19	7.278 hr

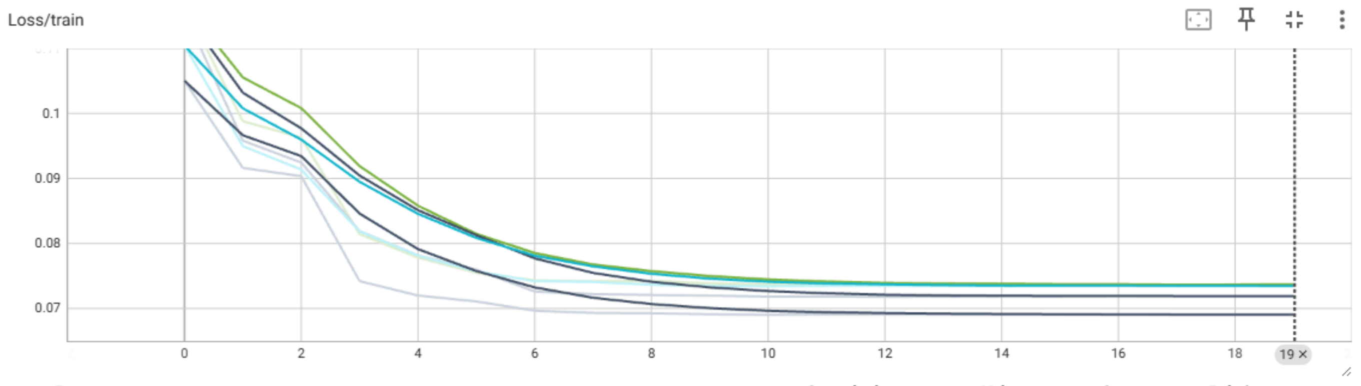
(b) Validation mAP



	Run ↑	Smoothed	Value	Step	Relative
●	resnet50_fpn_stLR3_0.1_nms0.3_b4_e20	0.9608	0.9608	19	7.332 hr
●	resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_A	0.8743	0.8755	11	4.133 hr
●	resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_binary	0.9184	0.9183	19	7.278 hr

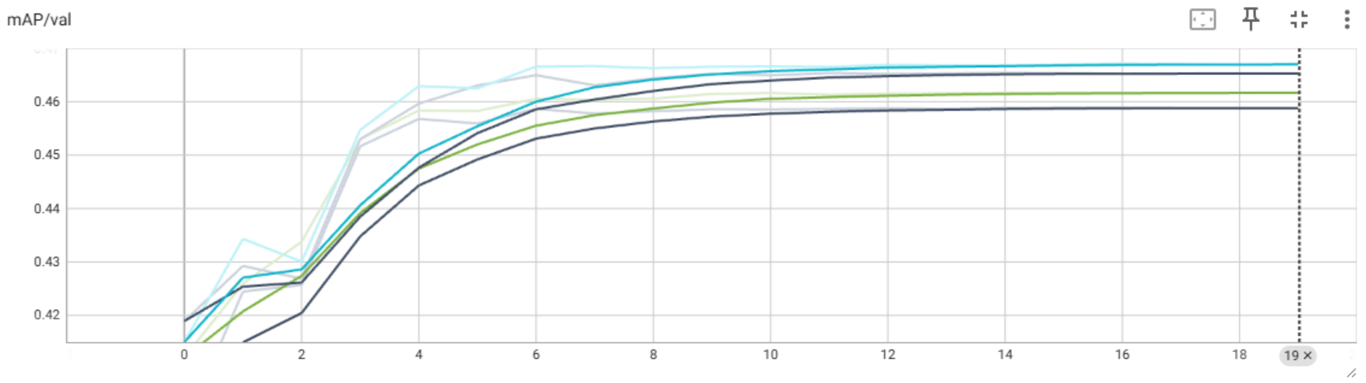
(c) Validation Accuracy

Figure 4: Overall performance comparison of data processing across training loss, mAP, and digit classification accuracy.



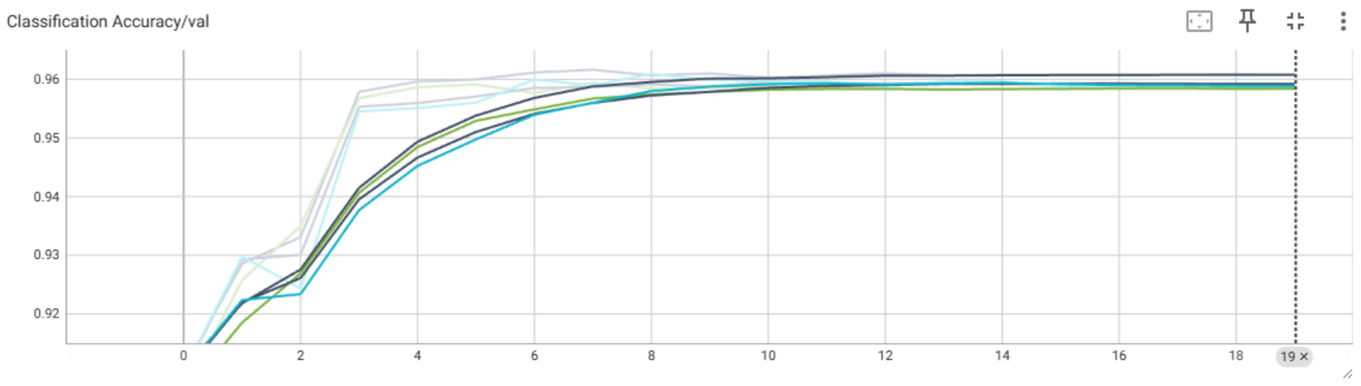
Run ↑	Smoothed	Value	Step	Relative
● resnet50_fpn_stLR3_0.1_nms0.3_b4_e20	0.069	0.069	19	7.334 hr
● resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_small_obj	0.0737	0.0737	19	7.827 hr
● resnet50_fpn_stLR3_0.1_nms0.3_b8_e20_small_obj_st0.5	0.0719	0.0719	19	7.097 hr
● resnet50_fpn_stLR3_0.1_nms0.3_b8_e20_small_obj_st0.5_ver	0.0735	0.0735	19	6.788 hr

(a) Training Loss



Run ↑	Smoothed	Value	Step	Relative
● resnet50_fpn_stLR3_0.1_nms0.3_b4_e20	0.4588	0.4588	19	7.332 hr
● resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_small_obj	0.4617	0.4617	19	7.827 hr
● resnet50_fpn_stLR3_0.1_nms0.3_b8_e20_small_obj_st0.5	0.4653	0.4653	19	7.088 hr
● resnet50_fpn_stLR3_0.1_nms0.3_b8_e20_small_obj_st0.5_ver	0.467	0.4671	19	6.787 hr

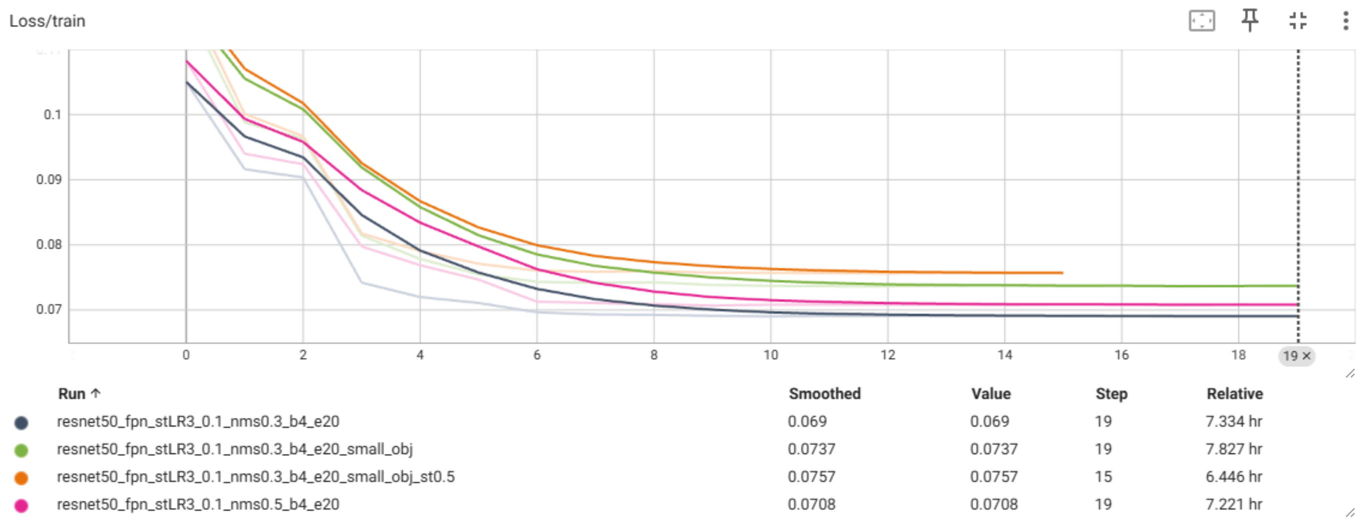
(b) Validation mAP



Run ↑	Smoothed	Value	Step	Relative
● resnet50_fpn_stLR3_0.1_nms0.3_b4_e20	0.9608	0.9608	19	7.332 hr
● resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_small_obj	0.9584	0.9584	19	7.827 hr
● resnet50_fpn_stLR3_0.1_nms0.3_b8_e20_small_obj_st0.5	0.9592	0.9592	19	7.088 hr
● resnet50_fpn_stLR3_0.1_nms0.3_b8_e20_small_obj_st0.5_ver	0.9589	0.9588	19	6.787 hr

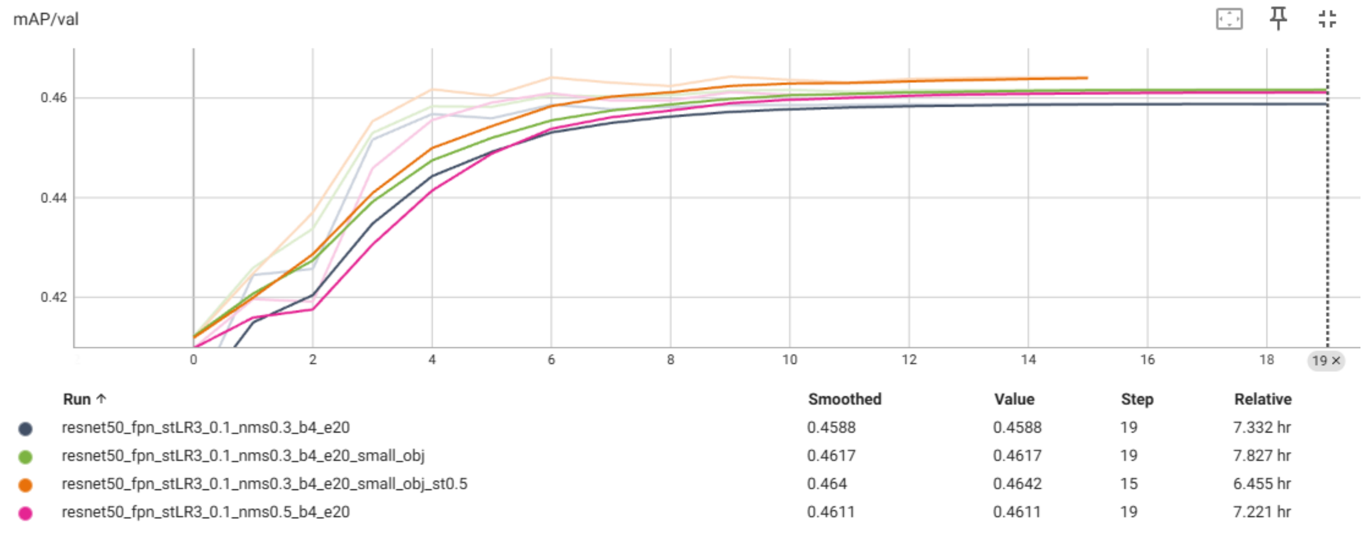
(c) Validation Accuracy

Figure 5: Overall performance comparison of different neck configurations across training loss, mAP, and digit classification accuracy.



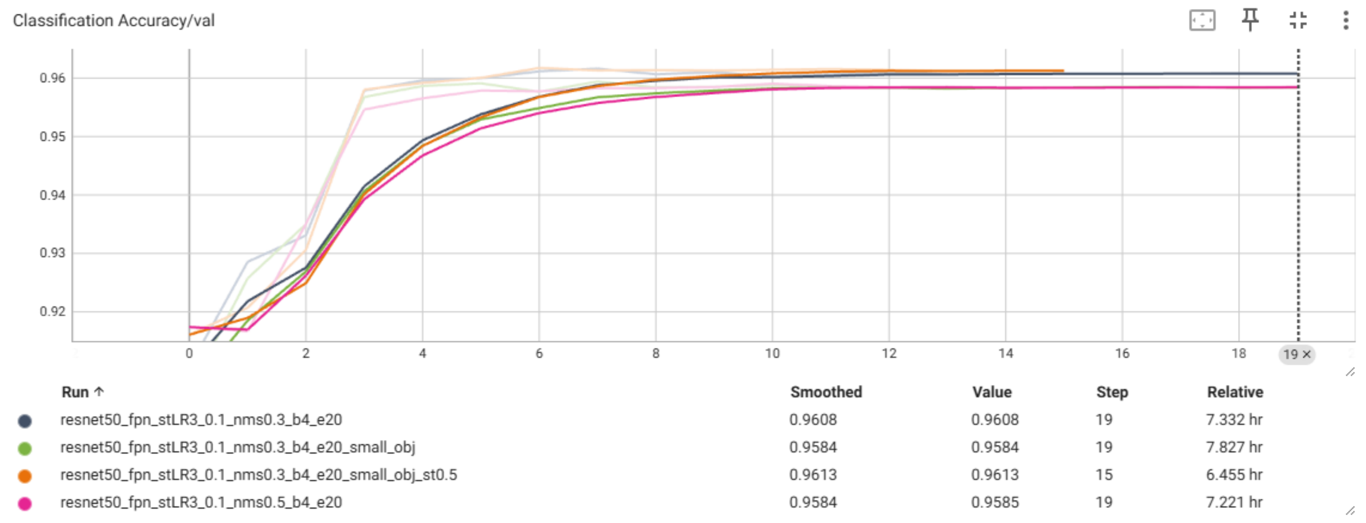
Run ↑	Smoothed	Value	Step	Relative
● resnet50_fpn_stLR3_0.1_nms0.3_b4_e20	0.069	0.069	19	7.334 hr
● resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_small_obj	0.0737	0.0737	19	7.827 hr
● resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_small_obj_st0.5	0.0757	0.0757	15	6.446 hr
● resnet50_fpn_stLR3_0.1_nms0.5_b4_e20	0.0708	0.0708	19	7.221 hr

(a) Training Loss



Run ↑	Smoothed	Value	Step	Relative
● resnet50_fpn_stLR3_0.1_nms0.3_b4_e20	0.4588	0.4588	19	7.332 hr
● resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_small_obj	0.4617	0.4617	19	7.827 hr
● resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_small_obj_st0.5	0.464	0.4642	15	6.455 hr
● resnet50_fpn_stLR3_0.1_nms0.5_b4_e20	0.4611	0.4611	19	7.221 hr

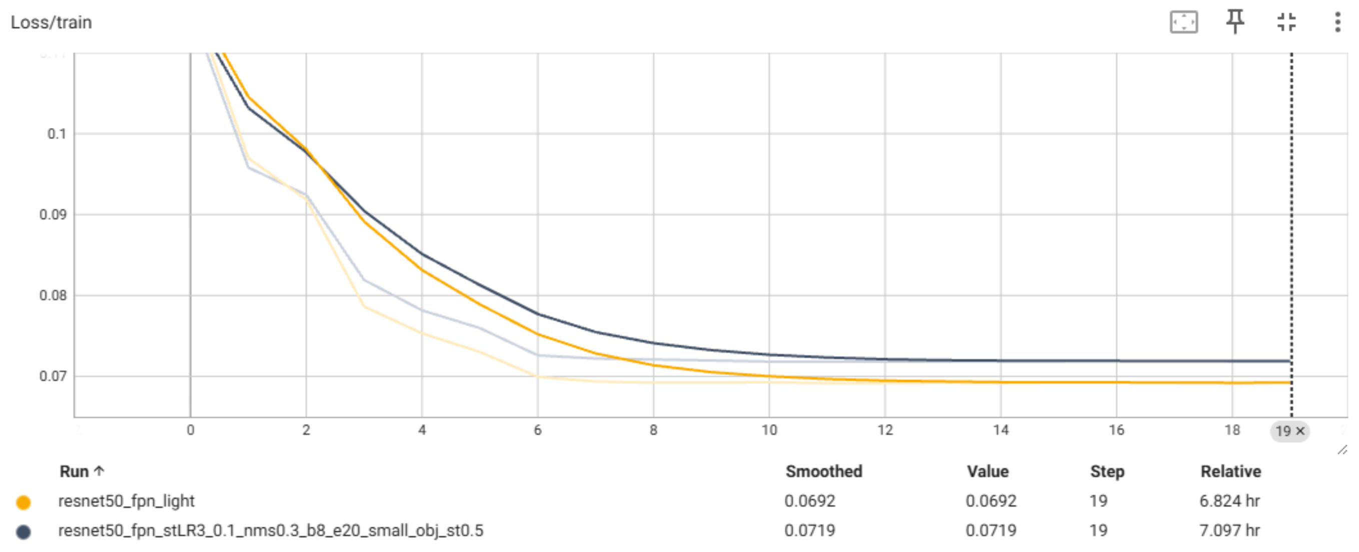
(b) Validation mAP



Run ↑	Smoothed	Value	Step	Relative
● resnet50_fpn_stLR3_0.1_nms0.3_b4_e20	0.9608	0.9608	19	7.332 hr
● resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_small_obj	0.9584	0.9584	19	7.827 hr
● resnet50_fpn_stLR3_0.1_nms0.3_b4_e20_small_obj_st0.5	0.9613	0.9613	15	6.455 hr
● resnet50_fpn_stLR3_0.1_nms0.5_b4_e20	0.9584	0.9585	19	7.221 hr

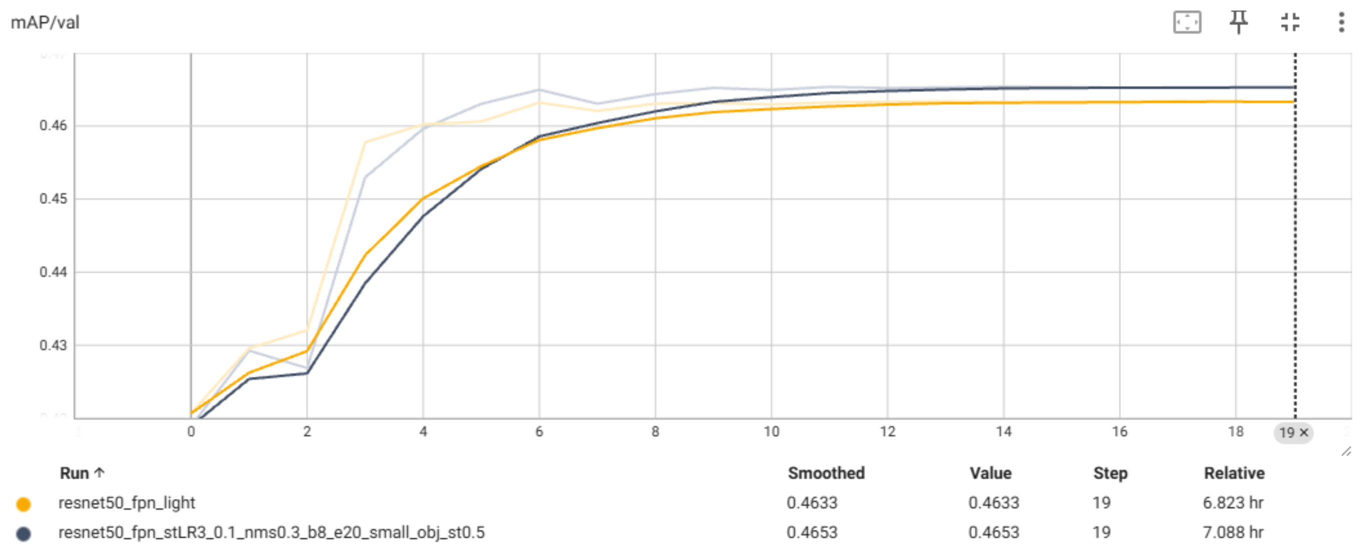
(c) Validation Accuracy

Figure 6: Overall performance comparison of NMS and score threshold settings across training loss, mAP, and digit classification accuracy.



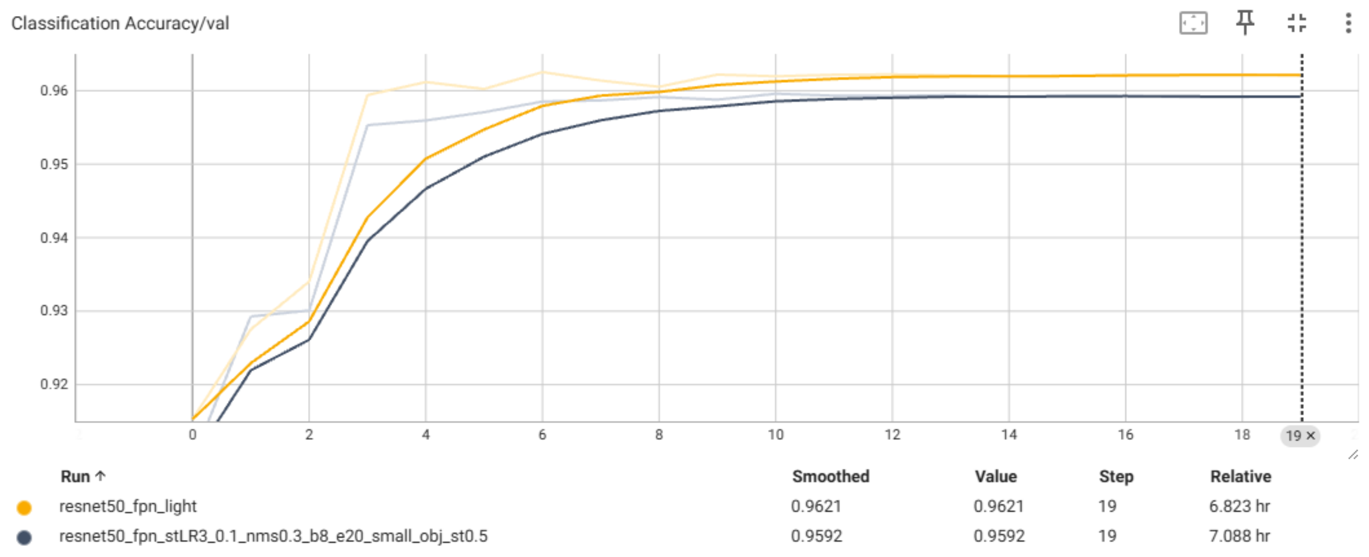
Run ↑	Smoothed	Value	Step	Relative
● resnet50_fpn_light	0.0692	0.0692	19	6.824 hr
● resnet50_fpn_stLR3_0.1_nms0.3_b8_e20_small_obj_st0.5	0.0719	0.0719	19	7.097 hr

(a) Training Loss



Run ↑	Smoothed	Value	Step	Relative
● resnet50_fpn_light	0.4633	0.4633	19	6.823 hr
● resnet50_fpn_stLR3_0.1_nms0.3_b8_e20_small_obj_st0.5	0.4653	0.4653	19	7.088 hr

(b) Validation mAP



Run ↑	Smoothed	Value	Step	Relative
● resnet50_fpn_light	0.9621	0.9621	19	6.823 hr
● resnet50_fpn_stLR3_0.1_nms0.3_b8_e20_small_obj_st0.5	0.9592	0.9592	19	7.088 hr

(c) Validation Accuracy

Figure 7: Overall performance comparison of different RoI pooling feature map configurations across training loss, validation mAP, and digit classification accuracy.

References

[1] R. Girshick, “Fast r-cnn,” 2015. [Online]. Available: https://arxiv.org/abs/1504.08083

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