Visual Recognition Using Deep Learning Lab2 多工所碩一 313553024 蘇柏叡

1. Introduction

In this lab, our objective is to perform object detection on a digit image dataset containing 10 classes (digits 0 through 9), and to output both the bounding box coordinates and predicted class labels. For the first task, we are required to perform inference on the test set using a pre-trained model and output the predictions in COCO format for the purpose of evaluating Mean Average Precision (mAP). The second task involves sorting the detected digits from left to right based on the x-axis coordinates of the predicted bounding boxes, and then computing the accuracy by comparing the resulting sequence with the ground truth annotations. Since the prediction results depend on a predefined confidence threshold during validation and inference, we initially evaluate the model using a default threshold of 0.5 to compute mAP.

Prior to final inference and result generation, we utilize a script named **Find_threshold.py** to evaluate a range of thresholds from 0.45 to 0.9 on the validation set. **For each threshold, we compute both the mAP for Task 1 and the accuracy for Task 2.** The threshold yielding the highest accuracy is then selected and integrated into the inference.py script, which subsequently generates the final outputs: pred.json for COCO evaluation and pred.csv for ordered digit accuracy assessment.

For model selection, **we adopt the Faster R-CNN architecture as a fixed framework**, while allowing modifications to the backbone, Feature Pyramid Network[5], Region Proposal Network (RPN), and the detection head. Specifically, for the backbone network, **we compare two pretrained variants provided by torchvision**: **fasterrcnn_resnet50_fpn[2] and fasterrcnn_resnet50_fpn_v2[3].** Due to hardware limitations and time constraints, we did not modify the RPN and head components, as the experiments in these areas could not be completed within the given timeframe. However, we conducted **additional experiments by incorporating a residual convolutional module into the existing FPN[5] structure** to explore potential performance improvements. Depending on time availability, we also plan to apply fine-tuning to the best-performing model weights in order to further enhance the overall detection accuracy and robustness of the model.

In the experimental and implementation stages, **our primary objective is to surpass both the weak and strong baselines.** To this end, we compare the performance of the two baseline models and **further investigate whether incorporating a residual module into the FPN[5] component of the superior model's backbone can yield improvements in overall model performance.** To enhance generalization and boost detection accuracy, we apply data augmentation techniques to the original training dataset prior to model training, thereby increasing data diversity. Given the strong resemblance between our dataset and the SVHN dataset, **we draw inspiration from SVHN-specific augmentation strategies[1]**, particularly those targeting color variations. These augmentations are adapted and fine-tuned accordingly. Additionally, we **incorporate a series of geometric and photometric transformations** — **including affine transformations, horizontal flipping, and ColorJitter—to further enrich the training data and improve model robustness.**

Regarding training strategies, we draw inspiration from the YOLOX[4] framework, which the author has previously studied and implemented. Accordingly, our training pipeline incorporates several techniques adapted from YOLOX[4]. Specifically, we adopt a **warm-up strategy** during the initial training phase to stabilize gradient updates, followed by **cosine annealing** to adjust the learning rate progressively throughout training. In the final training epochs, **we disable data augmentation to encourage convergence** and allow the model to better fit the true data distribution. These strategies, **combined with the aforementioned models(includes our proposed method)** and augmentation designs, **enabled us to successfully outperform both the weak and strong baselines across the two assigned tasks in this project.**

2. Method

2.1 Data Preprocessing

(1) Data Loader

In this lab, although the **input images vary in size** across the dataset, we **maintain their original resolutions** during training, validation, and testing to ensure consistency in evaluation and compatibility with mAP computation. Consequently, we do not perform any resizing operations on the images. For data preprocessing, the annotated bounding boxes **provided in COCO format are first converted to the [x_min, y_min, x_max, y_max] format.** These bounding boxes, along with their corresponding class labels, are then converted into PyTorch tensors to facilitate model training and inference in subsequent stages.

(2) Data Augmentation

For data augmentation, our implementation comprises the following four components:

a. ColorJitter:

We utilize **torchvision.transforms.ColorJitter** to adjust image properties such as brightness, contrast, and saturation. This augmentation is conditionally applied based on a predefined probability threshold associated with the HSV augmentation; specifically, the transformation is triggered when a randomly sampled value falls below the configured HSV probability.

b. Horizontal Flip:

Horizontal flipping is applied to images based on a predefined flip probability. It is important to note that such a transformation alters the spatial positioning of the objects within the image. Therefore, **the corresponding bounding boxes must be adjusted accordingly** by performing a **symmetric transformation** of the left and right coordinates with respect to the image width.

c. Affine Transformation:

For affine transformation, we set the **rotation angle to ±10°**, **apply random translations within 10% of the image dimensions** for both width and height, and define **shear parameters within the range of ±2.** Based on these parameters, an affine transformation matrix is constructed.

In addition to applying the transformation to the input image, special attention must be paid to the transformation of the corresponding bounding boxes. **Since the resulting boxes may no longer remain aligned with the original coordinate axes after rotation,** we apply the affine transformation to all four corner points of each bounding box. If the transformed box is no longer axis-aligned, we compute its **minimum enclosing rectangle** to ensure a proper rectangular bounding box is restored, and then update the coordinate annotations accordingly.

d. SubPolicy and SVHN Color Policy[1]:

This component is inspired by the method proposed in [1]. Specifically, the SubPolicy defines **a sequence of two consecutive color transformations**, each associated with a fixed probability of application and a predefined magnitude. These transformations may include adjustments to brightness, contrast, and color balance, thereby producing subtle yet diverse variations in image appearance.

The SVHNColorPolicy aggregates multiple SubPolicy instances, **focusing solely on color-based augmentation without altering geometric properties**. During training, one SubPolicy is randomly selected and applied to each image, thereby enhancing color diversity in the dataset and improving the model's robustness to color variations.

2.2 Model Architecture

In this study, we adopt two backbone architectures **from the torchvision library** as our baseline models. While these models are largely similar in their overall design, we begin by providing a general overview of the Faster R-CNN architecture. Subsequently, **we present a detailed comparison highlighting the architectural differences between fasterrcnn_resnet50_fpn[2] and fasterrcnn_resnet50_fpn_v2[3].**

(1) Backbone Network

In Faster R-CNN, the backbone network plays a crucial role in extracting hierarchical convolutional features from input images, which serve as the foundational input for the subsequent Region Proposal Network (RPN) and detection head. Common backbone architectures **include the ResNet, VGG, and MobileNet families**, which can be selected based on a balance between detection accuracy and computational efficiency.

In this study, we adopted the pretrained ResNet-50 with Feature Pyramid Network[5] as the backbone architecture. ResNet-50 is a deep convolutional neural network composed of 50 layers, characterized by its modular residual learning framework. Each **residual block incorporates a shortcut connection**, enabling the network to learn identity mappings and **effectively alleviating the vanishing gradient problem** frequently encountered in deep networks. This design facilitates improved convergence and expressive capacity. Through progressively deeper stages—namely conv2_x to conv5_x—ResNet-50 is capable of capturing multi-scale and increasingly abstract semantic features.

However, directly leveraging raw features from different stages—particularly through simple concatenation—**often results in semantic misalignment and information inconsistency**, which can hinder performance in detecting objects of varying sizes. To address this issue, Faster R-CNN integrates the FPN[5], which is specifically designed to enable multi-scale feature fusion.

FPN[5] introduces a top-down pathway complemented by multiple lateral connections. High-level semantic features with low resolution are upsampled and fused—typically via element-wise addition—with low-level features that have higher spatial resolution but weaker semantic content. The network uses 1×1 convolutions for channel alignment and 3×3 convolutions for refinement, preserving spatial details. This design **ensures consistent and semantically rich representations across all scales**, thereby enhancing the model's ability to detect objects of various sizes and aspect ratios.

(2) Modification of the Original FPN[5] and Its Integration into Additional Experiments

To further enhance the representational capacity of the Feature Pyramid Network[5], we propose an extended architecture **by appending an additional residual block after the FPN[5] outputs**, forming what we refer to as the Residual Feature Pyramid Network (ResFPN). Compared to directly using the original FPN[5] outputs, this design aims to improve the network’s non-linear transformation capacity while maintaining the multi-scale semantic structure. The residual block architecture, with its inherent skip connections, facilitates efficient information flow and gradient propagation in deeper layers, effectively addressing the vanishing gradient and degradation problems commonly encountered in deep networks. It also promotes more stable feature learning and improves the ability of the model to capture fine-grained spatial details and localized structures.

The motivation behind this modification is based on the observation that, **although FPN enables effective multi-scale feature fusion, its output depth remains limited**, particularly when dealing with small objects or regions containing intricate visual patterns. **By inserting a residual block after the FPN[5], we introduce an additional layer of high-level transformation that strengthens the semantic abstraction of the features before they are processed by RoI Align.** We expect this improvement to result in higher-quality region features and ultimately lead to better performance in both classification and bounding box regression tasks. Detailed experimental results and comparative evaluations are presented in Section 3.

(3) Region Proposal Network

In the Faster R-CNN architecture, the Region Proposal Network (RPN) is responsible for generating candidate object regions from the multi-scale feature maps produced by the backbone and the FPN[5]. These proposals are then forwarded to the detection head for final classification and bounding box refinement.

The RPN architecture begins with a shared 3×3 convolutional layer that extracts local features through sliding-window operations over the input feature maps, followed by a ReLU activation function to introduce non-linearity. It then **branches into two parallel sub-networks:** one performs **binary classification** to determine whether each anchor box corresponds to a foreground **object or background** (i.e., objectness score), while the other performs **bounding box regression** to adjust anchor coordinates for better alignment with the ground-truth object.

This fully convolutional and computationally efficient design enables the RPN to generate dense proposals across feature maps of varying resolutions. At each spatial location, the RPN produces multiple predefined anchor boxes, which are designed using a set of predefined scales and aspect ratios to effectively capture objects of different sizes and shapes. The candidate proposals are ranked based on their objectness scores, and highly overlapping boxes are suppressed using Non-Maximum Suppression (NMS) to retain only the most relevant regions.

When integrated with the Feature Pyramid Network[5], the RPN operates independently on each scale of the feature hierarchy. The proposals generated at each level are then aggregated and subjected to a final round of NMS to produce the final set of object proposals. This multi-scale strategy significantly improves the model's robustness and accuracy in detecting objects across a wide range of sizes and resolutions.

(4) Detection Head

In the Faster R-CNN architecture, the detection head is the final component responsible for processing the candidate regions proposed by the RPN. It performs the tasks of object classification and bounding box regression. The detection process can be divided into two main stages.

In the first stage, **RoI Align is employed to extract fixed-size feature representations** from each proposed region across the multi-scale feature maps. This approach **avoids the quantization artifacts introduced by traditional RoI Pooling**, thereby preserving spatial alignment and ensuring consistent input dimensions for subsequent layers.

In the second stage, the fixed-size RoI features are **flattened into one-dimensional vectors** and passed through fully connected layers to extract high-level semantic representations. These are then split into two parallel branches: a **classification branch**, which uses a softmax layer to predict the probability distribution over object classes (including the background), and a **regression branch**, which outputs the bounding box offsets to refine the proposals for better alignment with the ground truth.

Both branches share the same RoI features extracted via RoI Align, and are trained jointly under a **multi-task learning** framework. This design enables simultaneous optimization of classification and localization objectives, thereby improving the overall accuracy and robustness of the object detection pipeline.

(5) Difference between fasterrcnn_resnet50_fpn and fasterrcnn_resnet50_fpn_v2

a. FPN Batch Normalization Settings:

i. fasterrcnn_resnet50_fpn[2]:

This version **employs frozen Batch Normalization layers**, which remain fixed throughout the training process. As a result, the parameters within these BN layers are not updated during training, thereby limiting the ability of the model to dynamically adapt to shifts in the data distribution.

ii. fasterrcnn_resnet50_fpn_v2[3]:

In the updated architecture, **Batch Normalization layers within the Feature Pyramid Network[5] are fully activated and no longer frozen.** This design choice enables the network to dynamically **update the running mean and variance statistics** during feature propagation, thereby improving its adaptability and robustness to varying batch compositions and changes in input image distributions.

b. Region Proposal Network (RPN) Architecture

i. fasterrcnn_resnet50_fpn[2]:

In the baseline version, the Region Proposal Network (RPN) employs a single 3×3 convolutional layer that performs sliding-window operations over the feature maps. This layer simultaneously predicts objectness scores and bounding box coordinates. Due to the simplicity of **using a single convolutional layer**, architectural adjustments were introduced in the v2 version to enhance the RPN's representational capacity and performance.

ii. fasterrcnn_resnet50_fpn_v2[3]:

In this version, **an additional 3×3 convolutional layer** is introduced to the original Region Proposal Network (RPN) architecture, thereby increasing the overall network depth. The motivation behind this design is to enhance the non-linear representation capacity through the stacking of two convolutional layers, which is expected to improve the network’s ability to distinguish candidate regions of varying scales and aspect ratios. The remaining operations follow the original design, where sliding windows are applied across the feature maps to simultaneously predict objectness scores and bounding box coordinates.

c. Box Regression Head Design

i. fasterrcnn_resnet50_fpn[2]:

In the baseline version, the detection head responsible for bounding box regression and classification is primarily composed of **two fully connected layers (MLPs)**, which operate on feature vectors extracted via RoI Align. However, as MLPs require flattening the RoI feature maps prior to processing, this approach tends to discard the spatial structure and local details within each region, which are crucial for precise bounding box refinement.

Moreover, **Batch Normalization is not applied within the fully connected layers**, thereby limiting the model's ability to dynamically adapt to changes in data distribution during training. To address these limitations, the v2 version abandons the use of MLPs in favor of an alternative head design.

ii. fasterrcnn_resnet50_fpn_v2[3]:

In the updated version, the box regression head is redesigned to consist of **four convolutional layers** followed by a single linear layer. The convolutional layers operate directly on the RoI feature maps, preserving their spatial structure while enabling the extraction of richer semantic information and local details through deeper hierarchical representations.

Each convolutional layer is followed by a Batch Normalization layer, which helps to dynamically stabilize activation distributions and **mitigate the risk of gradient instability during training**. After the stacked convolutional operations yield high-dimensional features with preserved spatial context, a fully connected layer is applied to perform the final mapping and decision-making, producing both classification logits and bounding box regression outputs.

2.3 Hyperparameters Settings and Training Configurations

2.3.1 Hyperparameters Settings

All experiments in this study were conducted on an **NVIDIA RTX 4060 Ti GPU with 16 GB of VRAM.** Due to limitations in computational resources and training time, we adjusted key training configurations such as batch size and the number of training epochs. Specifically, when using fasterrcnn_resnet50_fpn_v2[3] as the backbone, we found that a batch size of 4 led to excessive training durations, making it impractical within our time constraints. To address this, the batch size was reduced to 2, and the total number of epochs was set to 40 to ensure sufficient training while maintaining manageable runtime.

For the additional experiment involving our proposed modification—integrating a Residual FPN module into the fasterrcnn_resnet50_fpn_v2[3] architecture—the increased model complexity further extended the training time. Consequently, we limited the number of training epochs to 30. In line with the main experiments, data augmentation was disabled during the final 15 epochs to promote convergence.

Regarding learning rate scheduling, **we adopted a cosine annealing strategy.** Additionally, a layer-wise learning rate scheme was implemented: the detection head was trained with the full initial learning rate, **while the backbone was updated using a scaled-down rate (initial learning rate × 0.5)**. This strategy aimed to enable the detection-specific components to adapt more quickly to new data, while preserving stable learning dynamics within the pretrained feature extraction backbone. The detailed training configurations for the two pretrained backbones, the Residual FPN experiment, and fine-tuning settings are summarized below.

(1) fasterrcnn_resnet50_fpn[2]:

- Batch size: 8
- Learning Rate: 1e-4
- Minimum Learning Rate: 5e-6
- Epoch: 50
- Warmup Epochs: 5
- Weight Decay: 5e-4
- Optimizer: Adam

(2) **fasterrcnn_resnet50_fpn_v2[3]:**

- Batch size: **2**
- Learning Rate: 1e-4
- Minimum Learning Rate: 5e-6
- Epoch: 40
- Warmup Epochs: 5
- Weight Decay: 5e-4
- Optimizer: Adam

(3) **fasterrcnn_resnet50_fpn_v2[3] with Residual FPN which we proposed:**

- Batch size: **2**
- Learning rate: 1e-4
- Minimum Learning Rate: 5e-6
- Epoch: 30
- Warmup Epochs: 5
- Weight Deca: 5e-4
- Optimizer: Adam

(4) **Fine-Tuning:**

- **Batch size will be the same as the model which you select.**
- Learning Rate: 5e-6
- Minimum Learning Rate: 5e-8
- Epoch: 20
- Warmup Epochs: 1 (In order to keep the initial learning rate)
- Weight Decay: 5e-4
- Optimizer: Adam

2.3.2 Training Configurations

(1) Data Augmentation and Application Strategy

As described in the Introduction and Section 2.1, various data augmentation techniques were applied to the training set to enhance data diversity. However, these augmentations were not employed throughout the entire training process. Instead, **all augmentation operations were disabled during the final 15 epochs** to facilitate more stable convergence and allow the model to better fit the underlying data distribution in the later stages of training.

(2) Warmup and Cosine Annealing

We set the number of warm-up epochs to five, during which the learning rate increases linearly from zero to the predefined initial learning rate. Following the warm-up phase, the learning rate is scheduled to decay according to a cosine annealing strategy, gradually decreasing from the initial value to a predefined minimum learning rate (eta_min) by the end of the specified total number of training epochs.

(3) Calculation of the Loss Function

The loss function utilized in this experiment comprises the following four components:

(1) **Classification loss**

(2) **Bounding box regression loss**

(3) **Objectness loss of the Region Proposal Network (RPN)**

(4) **Bounding box regression loss of the RPN**.

Specifically, the **classification loss** is responsible for assigning a class label to each Region of Interest (RoI) and is computed using the cross-entropy loss function. The **bounding box regression loss** is calculated using the Smooth L1 loss, which measures the difference between the predicted bounding boxes and the ground truth boxes.

The **objectness loss** of the RPN determines whether a given anchor corresponds to a foreground object or background, while the **RPN bounding box regression loss** quantifies the localization error between the anchor boxes and the predicted proposals.

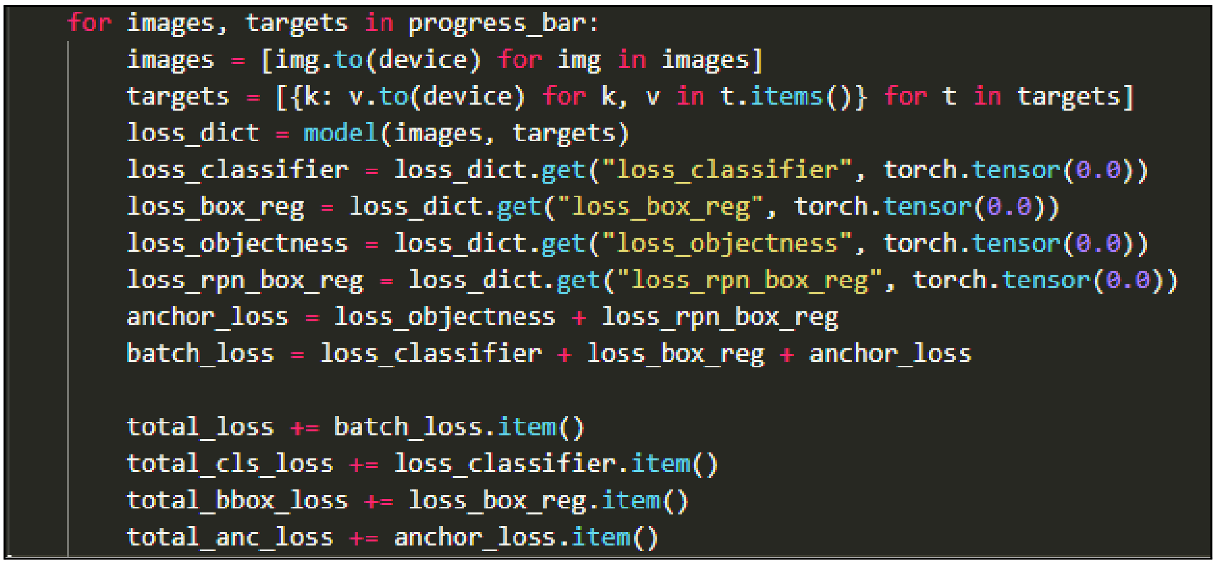
During training, these four components are summed to form the total loss. For monitoring and interpretability purposes, we further group them into three aggregated categories:

(1) Cls_loss represents the classification loss.

(2) Bbox_loss represents the bounding box regression loss.

(3) Anc_loss represents the combination of RPN objectness loss and RPN regression loss.

This categorization allows for clearer insight into which components contribute most significantly to the overall loss during training, and helps identify potential bottlenecks in the model's learning process.



```
for images, targets in progress_bar:
    images = [img.to(device) for img in images]
    targets = [{k: v.to(device) for k, v in t.items()} for t in targets]
    loss_dict = model(images, targets)
    loss_classifier = loss_dict.get("loss_classifier", torch.tensor(0.0))
    loss_box_reg = loss_dict.get("loss_box_reg", torch.tensor(0.0))
    loss_objectness = loss_dict.get("loss_objectness", torch.tensor(0.0))
    loss_rpn_box_reg = loss_dict.get("loss_rpn_box_reg", torch.tensor(0.0))
    anchor_loss = loss_objectness + loss_rpn_box_reg
    batch_loss = loss_classifier + loss_box_reg + anchor_loss

    total_loss += batch_loss.item()
    total_cls_loss += loss_classifier.item()
    total_bbox_loss += loss_box_reg.item()
    total_anc_loss += anchor_loss.item()
```

Fig 1. Code of the respective part of the Total Loss Function

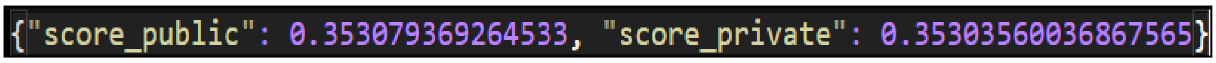
3. Results & Additional experiments

3.1 The baseline experiment and additional experiment results

(1) Using fasterrcnn_resnet50_fpn[2] as our model

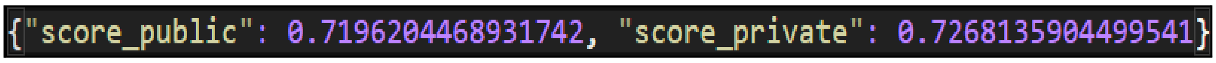
In this experiment, we followed the hyperparameter settings described in the previous section. The loss curves show no signs of overfitting, with both training and validation losses converging around 0.23. Although moderate fluctuations were observed during mid-training, the overall trend remained steadily convergent. After data augmentation was disabled at **Epoch 36, the loss curve became noticeably smoother**, **and the validation mAP improved significantly around Epoch 40**, reaching approximately 0.4449 with a threshold of 0.5.

Using Find_threshold.py, we determined that a threshold of 0.7 yielded the highest validation accuracy, and applied it during final inference via inference.py. Upon submission to CodaBench, our model achieved 0.3530 mAP for Task 1 and 0.7268 accuracy for Task 2—both surpassing the strong baseline benchmarks of 0.35 and 0.70, respectively.



```
{"score_public": 0.353079369264533, "score_private": 0.35303560036867565}
```

Fig 2. The plot of the mAP score using fasterrcnn_resnet50_fpn[2] in Task 1.



```
{"score_public": 0.7196204468931742, "score_private": 0.7268135904499541}
```

Fig 3. The plot of the Accuracy score using fasterrcnn_resnet50_fpn[2] in Task 2.

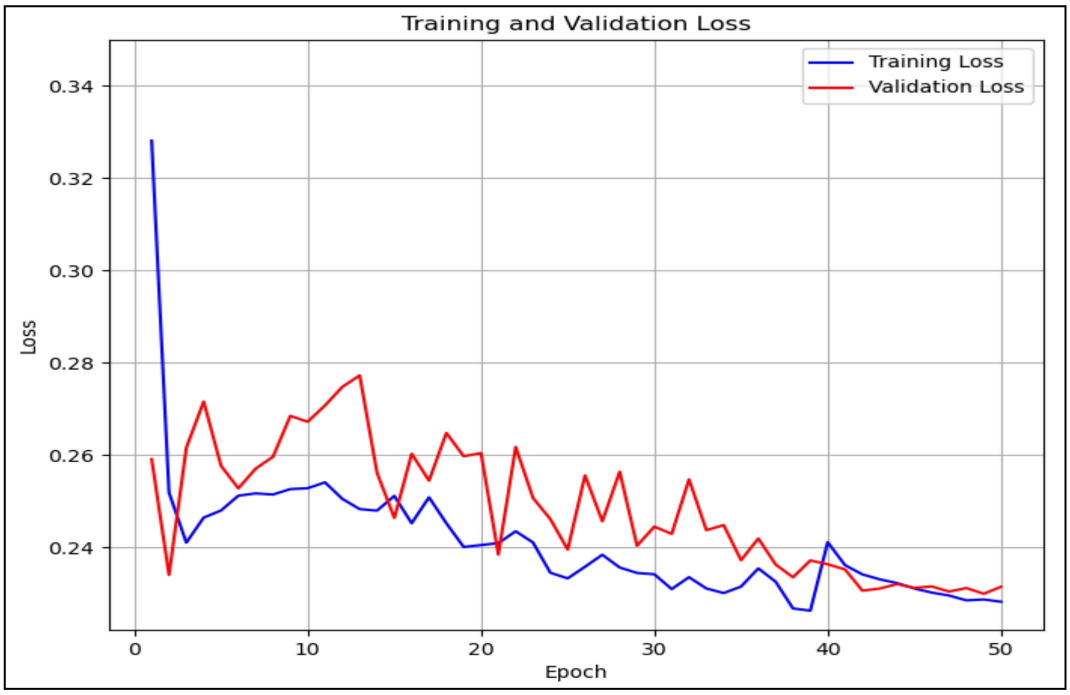


Fig 4. The plot of Training and Validation Loss with using fasterrcnn_resnet50_fpn[2]

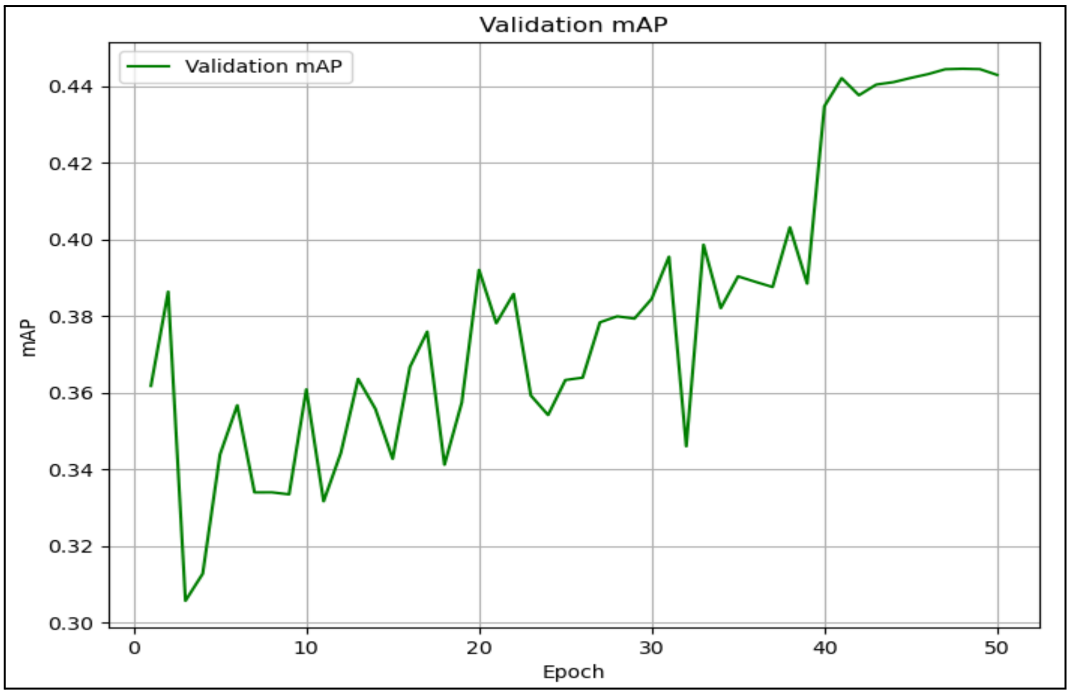
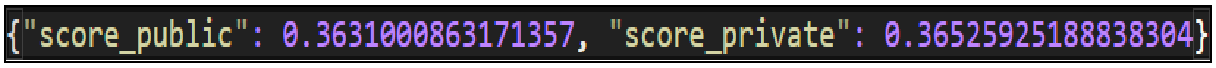


Fig 5. The plot of Validation mAP with using fasterrcnn_resnet50_fpn[2]

(2) Using fasterrcnn_resnet50_fpn_v2[3] as our model

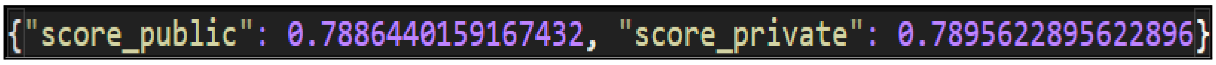
In this experiment, we adopted the same hyperparameter settings described in the previous section. **The training loss converged to below 0.16, while the validation loss stabilized around 0.18**. However, it is worth noting that signs of overfitting began to emerge during the final stages of training. **Therefore, for future fine-tuning, increasing the weight decay value or using ReduceLROnPlateau scheduler and Early Stopping mechanism may help mitigate this issue and improve fine-tuning effectiveness.** Although both mAP and loss exhibited fluctuations during the early training epochs, the overall trend indicated gradual convergence. From Epoch 26—when data augmentation was disabled—the loss curve became noticeably smoother and more stable. The model eventually achieved a validation mAP of approximately 0.4502 with a threshold of 0.5.

Using the Find_threshold.py utility, we identified a threshold of 0.79 as yielding the highest validation accuracy. This threshold was subsequently adopted in the final inference phase via inference.py. Upon submission to CodaBench, our model achieved an mAP of **0.3653 for Task 1 and an accuracy of 0.7896 for Task 2.** These results not only surpass the strong baseline benchmarks of 0.35 mAP and 0.70 accuracy, but also **outperform the previous results** obtained using fasterrcnn_resnet50_fpn[2] **(0.3530 mAP and 0.7268 accuracy).**



```
{"score_public": 0.3631000863171357, "score_private": 0.36525925188838304}
```

Fig 6. The plot of the mAP score using fasterrcnn_resnet50_fpn_v2[3] in Task 1.



```
{"score_public": 0.7886440159167432, "score_private": 0.7895622895622896}
```

Fig 7. The plot of the Accuracy score using fasterrcnn_resnet50_fpn_v2[3] in Task 2.

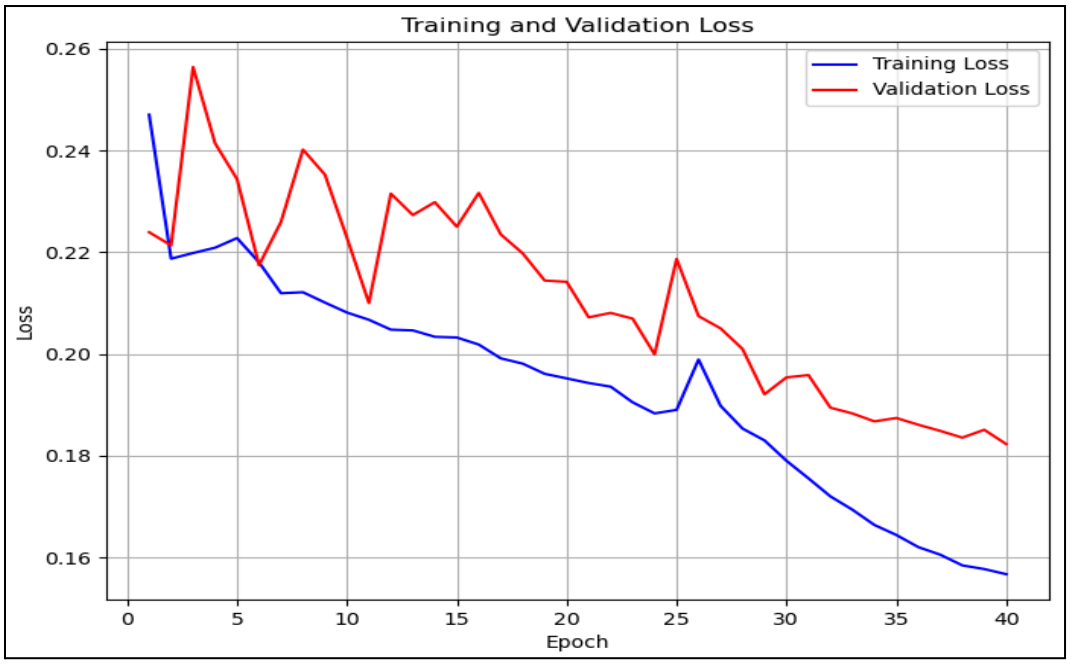


Fig 8. The plot of Training and Validation Loss with using fasterrcnn_resnet50_fpn_v2[3]

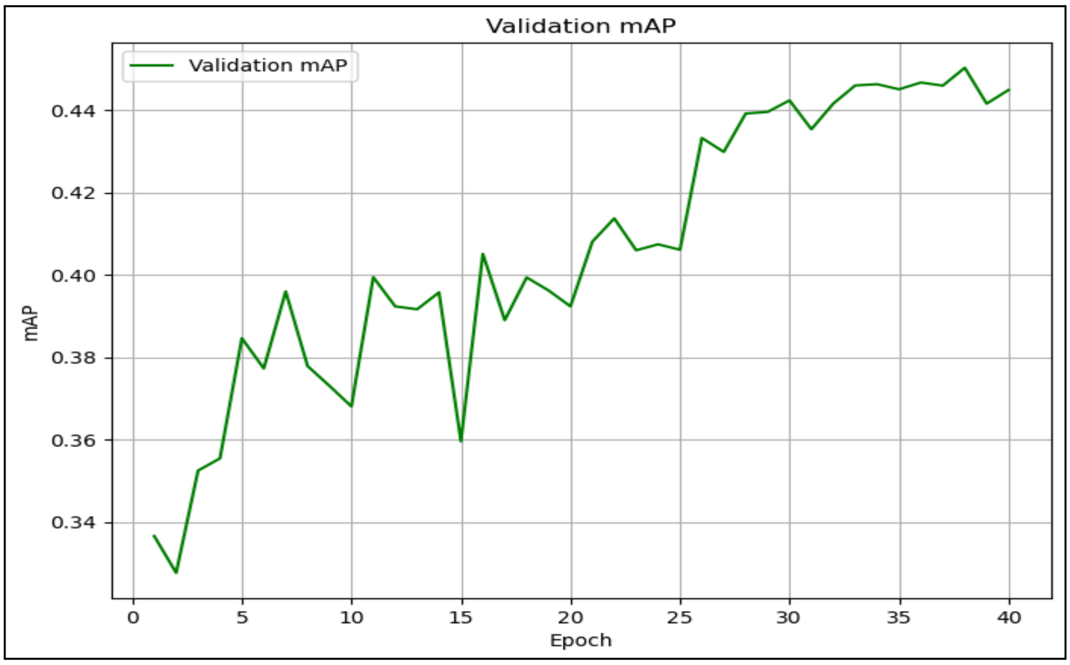


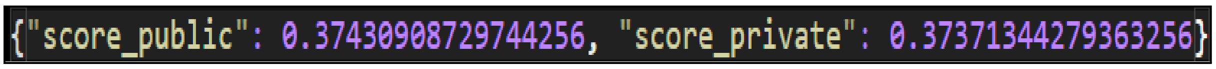
Fig 9. The plot of Validation mAP with using fasterrcnn_resnet50_fpn_v2[3]

(3) Using fasterrcnn_resnet50_fpn_v2[3] with Residual FPN which we proposed as our additional experiment

In this experiment, we adopted the same hyperparameter configuration described in the previous section. **The training loss converged to approximately 0.17, while the validation loss stabilized around 0.20. However, signs of potential overfitting began to emerge toward the later stages of training.** To mitigate this issue, it may be beneficial to reduce the number of epochs without data augmentation when performing fine-tuning, in order to avoid performance degradation.

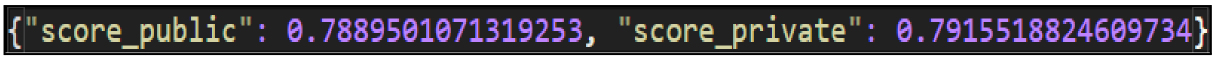
Notably, we observed that starting from Epoch 16, when data augmentation was disabled, the loss curve exhibited a smoother and more stable convergence pattern. At the end of training, the model achieved a mean Average Precision (mAP) of 0.4531 on the validation set with a classification threshold of 0.5.

Subsequently, we used the Find_threshold.py utility to determine the optimal confidence threshold, identifying 0.71 as the value that yielded the highest validation accuracy. This threshold was then applied in inference.py for final evaluation. Upon submission to CodaBench, the model **attained an mAP of 0.3737 for Task 1 and an accuracy of 0.7916 for Task 2.** These results not only surpassed the strong baseline benchmarks of 0.35 mAP and 0.70 accuracy, **but also outperformed the results obtained using fasterrcnn_resnet50_fpn_v2[3], which achieved 0.3653 mAP and 0.7896 accuracy, respectively.**



```
{"score_public": 0.37430908729744256, "score_private": 0.37371344279363256}
```

Fig 10. The plot of the mAP score using fasterrcnn_resnet50_fpn_v2[3] with Residual FPN which we proposed in Task 1.



```
{"score_public": 0.7889501071319253, "score_private": 0.7915518824609734}
```

Fig 11. The plot of the Accuracy score using fasterrcnn_resnet50_fpn_v2[3] with Residual FPN which we proposed in Task 2.

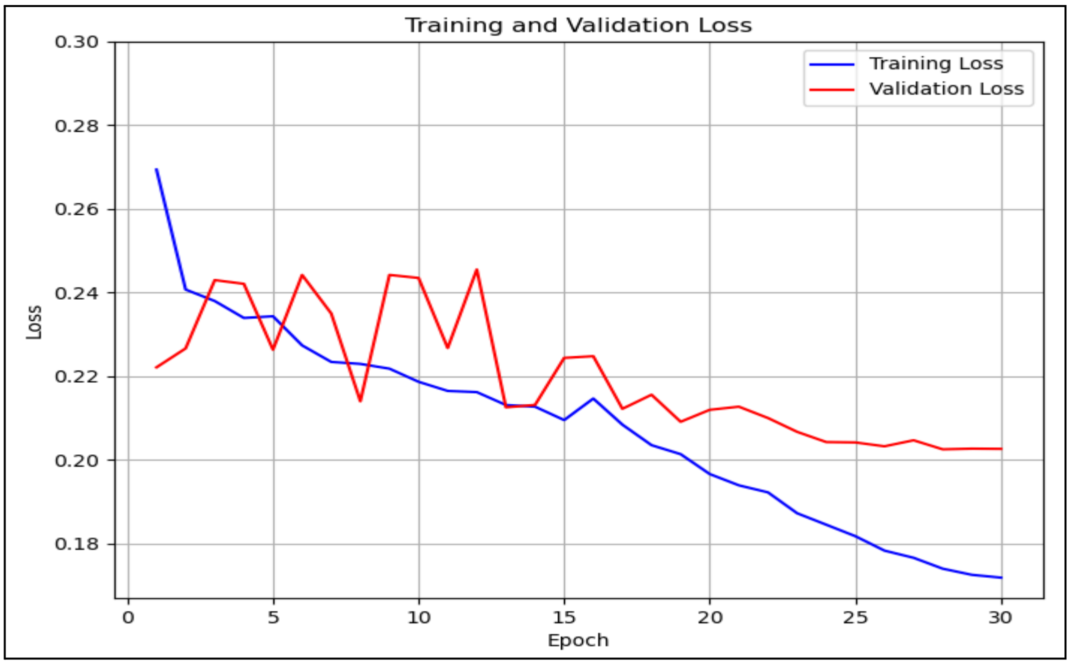


Fig 12. The plot of Training and Validation Loss with using fasterrcnn_resnet50_fpn_v2[3] with Residual FPN which we proposed

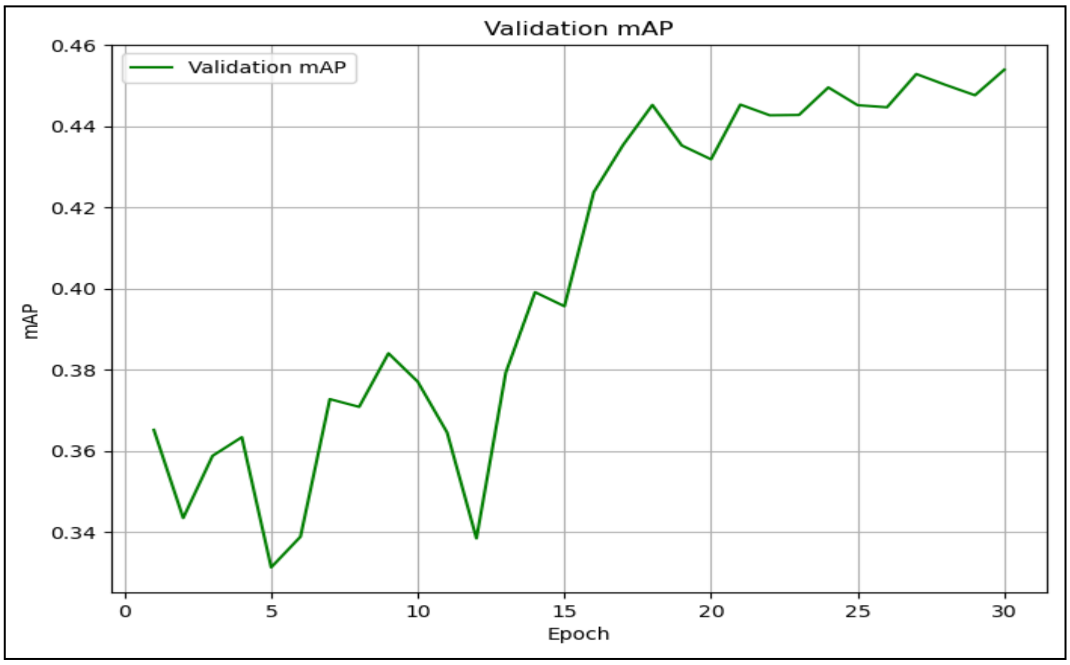


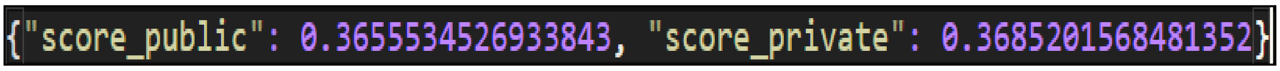
Fig 13. The plot of Validation mAP with using fasterrcnn_resnet50_fpn_v2[3]

3.2 Fine-tuning results

(1) Fine-Tuning fasterrcnn_resnet50_fpn[2]

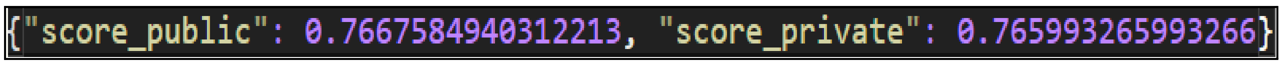
In this experiment, we adopted the fine-tuning configuration described in Section 2.3. Based on the loss and validation mAP curves, we observed a slight reduction in the loss value following fine-tuning, though potential signs of overfitting also began to emerge. Nevertheless, the validation mAP improved from a previous peak of 0.4449 to 0.4603, with the classification threshold set to 0.5.

Using the Find_threshold.py utility, we identified 0.72 as the optimal threshold and applied it in inference.py. Upon submission, the model achieved an **mAP of 0.3685 for Task 1 and an accuracy of 0.7660 for Task 2.** These results not only surpassed the strong baseline benchmarks of 0.35 mAP and 0.70 accuracy, but also **outperformed the non-fine-tuned fasterrcnn_resnet50_fpn[2] model (0.3530 mAP, 0.7268 accuracy), yielding gains of approximately 0.0155 in mAP and 0.0392 in accuracy.**



```
{"score_public": 0.3655534526933843, "score_private": 0.3685201568481352}
```

Fig 14. The plot of the mAP score Fine-Tuning fasterrcnn_resnet50_fpn[2] in Task 1.



```
{"score_public": 0.7667584940312213, "score_private": 0.765993265993266}
```

Fig 15. The plot of the Accuracy score Fine-Tuning fasterrcnn_resnet50_fpn[2] in Task 2.

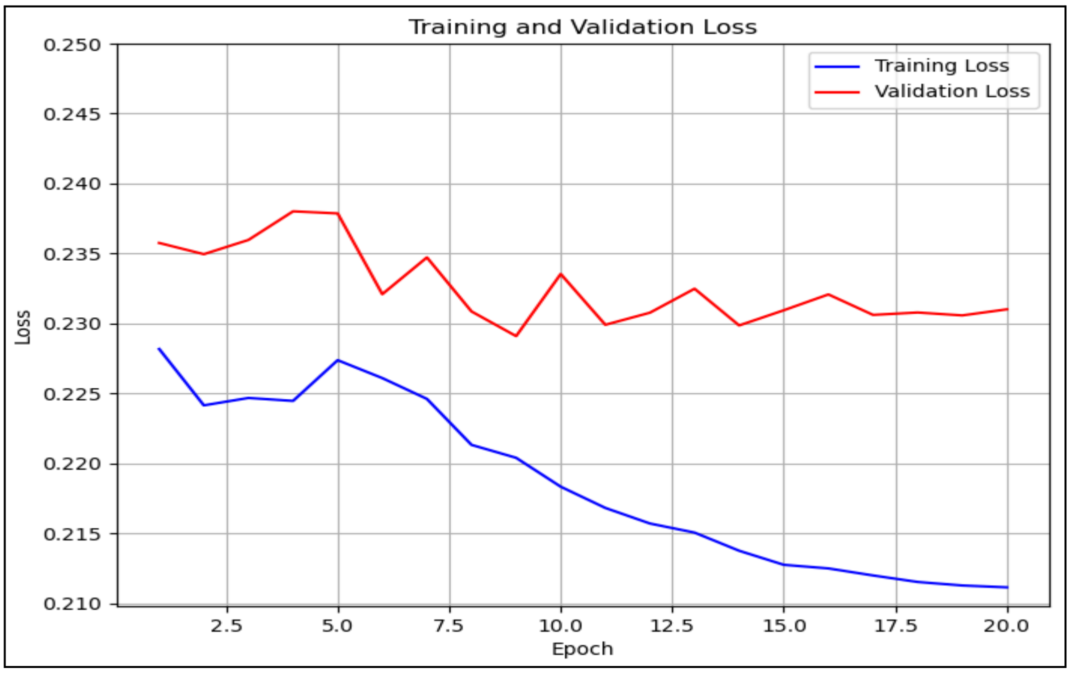


Fig 16. The plot of Training and Validation Loss with Fine-Tuning fasterrcnn_resnet50_fpn[2]

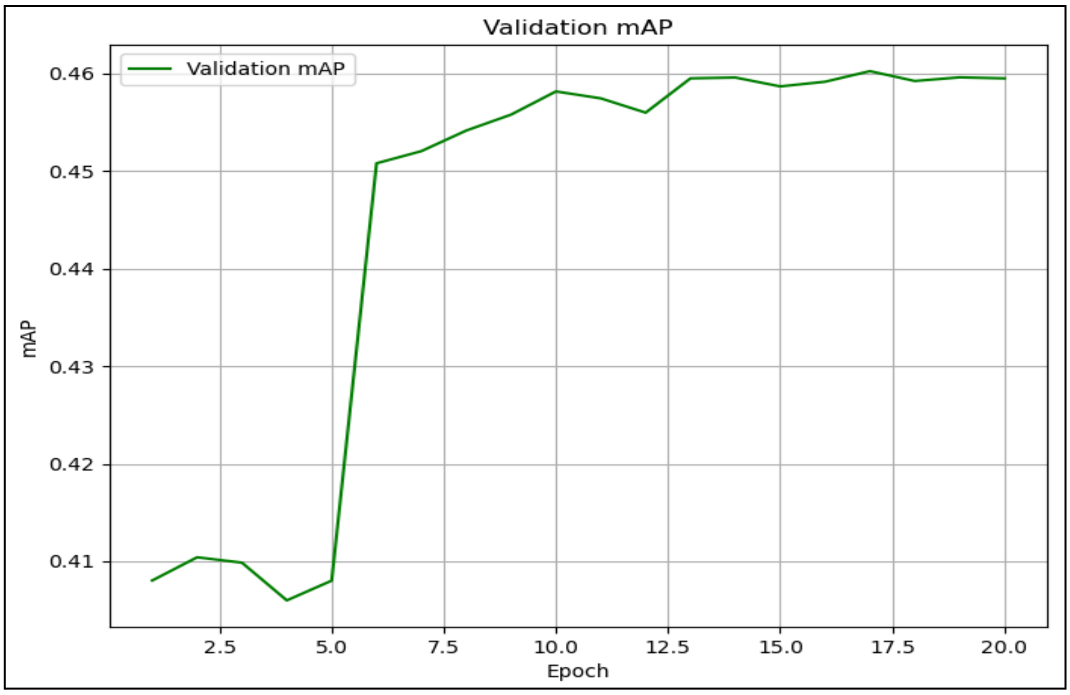
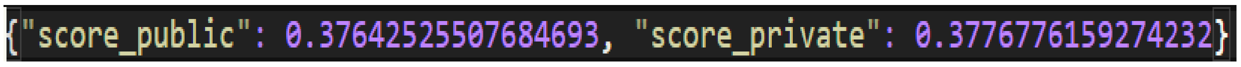


Fig 17. The plot of Validation mAP with Fine-Tuning fasterrcnn_resnet50_fpn[2]

(2) Fine-Tuning fasterrcnn_resnet50_fpn_v2[3]

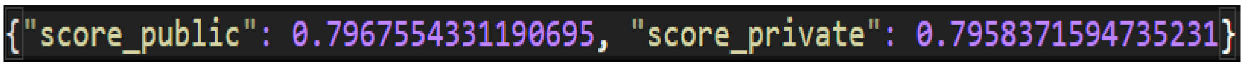
In this experiment, we made slight adjustments to the fine-tuning configuration described in Section 2.3 **by reducing the number of epochs without data augmentation from 15 to 12.** Based on the training loss and validation mAP curves, we observed **a slight decrease in loss values after fine-tuning compared to the original training phase.** However, the loss trajectory during fine-tuning suggests potential signs of overfitting. Despite this, the validation mAP curve showed a marginal improvement, increasing from a previous peak of 0.4531 to 0.4554 when evaluated with a threshold of 0.5. Prior to submission to CodaBench, we applied Find_threshold.py to determine the optimal confidence threshold. The highest accuracy was achieved at a threshold of 0.71, which was subsequently used in inference.py.

The final submission to CodaBench yielded an mAP of 0.3777 for Task 1 and an accuracy of 0.7958 for Task 2. These results exceeded both the strong baseline benchmarks (0.35 mAP, 0.70 accuracy) and the performance of the non–fine-tuned **fasterrcnn_resnet50_fpn_v2 model[3] with Residual FPN (0.3737 mAP, 0.7916 accuracy), reflecting relative improvements of 0.40% in mAP and 0.42% in accuracy.** Due to the fact that only one day remained before the submission deadline upon completion of this fine-tuning experiment, we were unable to conduct additional trials to further investigate the observed overfitting behavior. Nevertheless, based on our analysis of the loss and validation curves, we hypothesize that the adoption of a more adaptive learning rate schedule—**such as ReduceLROnPlateau—in conjunction with an early stopping mechanism may help mitigate overfitting in future experiments.** These techniques could potentially improve model generalization by dynamically adjusting the learning process and halting training once performance on the validation set plateaus.



```
{"score_public": 0.37642525507684693, "score_private": 0.3776776159274232}
```

Fig 18. The plot of the mAP score Fine-Tuning fasterrcnn_resnet50_fpn[2] with Residual FPN which we proposed in Task 1.



```
{"score_public": 0.7967554331190695, "score_private": 0.7958371594735231}
```

Fig 19. The plot of the Accuracy score Fine-Tuning fasterrcnn_resnet50_fpn[2] with Residual FPN which we proposed in Task 2.

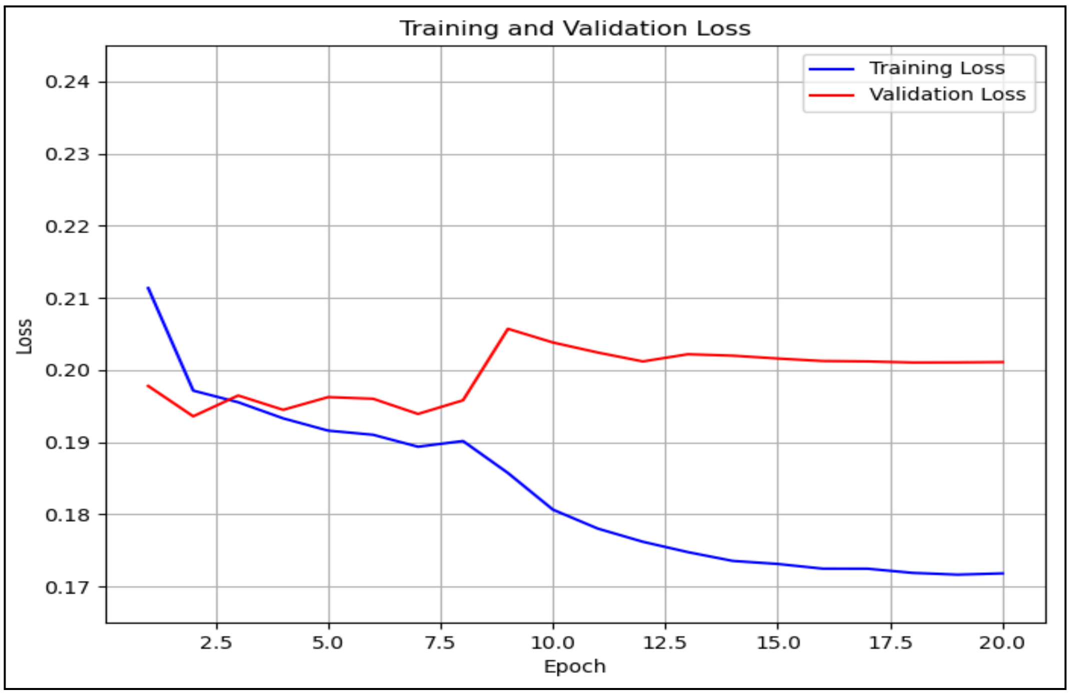


Fig 16. The plot of Training and Validation Loss with Fine-Tuning fasterrcnn_resnet50_fpn[2] with Residual FPN which we proposed

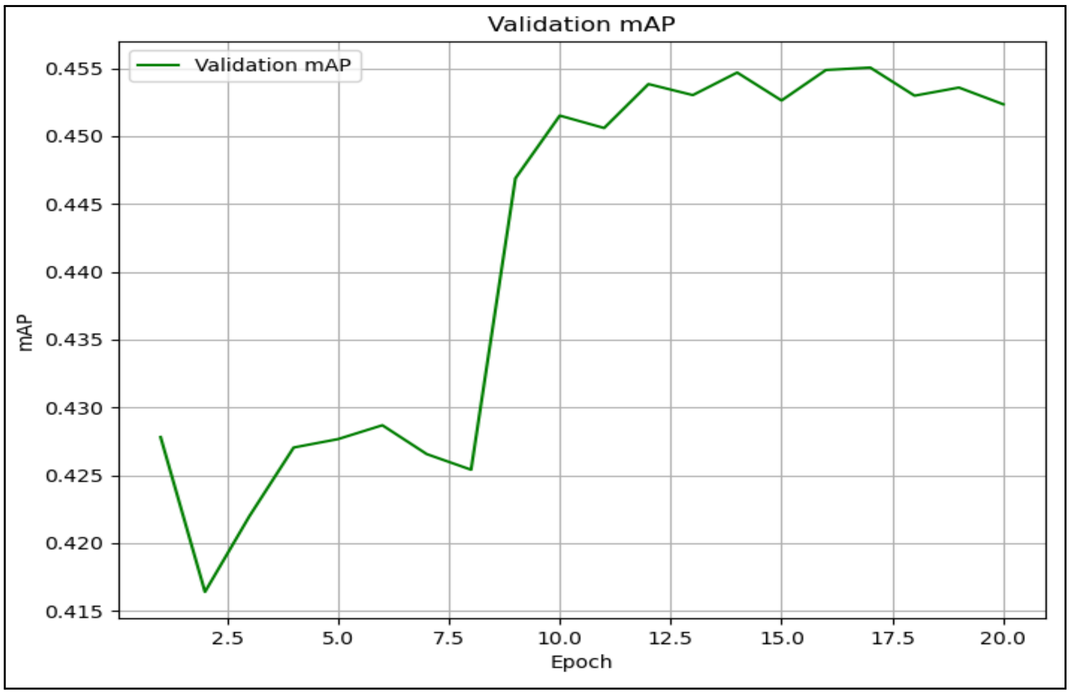


Fig 17. The plot of Validation mAP with Fine-Tuning fasterrcnn_resnet50_fpn[2] with Residual FPN which we proposed

4. Discussion and Conclusion

4.1 Experiment Results and Analysis

Model Name	val mAP	val Accuracy	test mAP	test Accuracy
fasterrcnn_resnet50_fpn[2] o/ Fine-Tuning	0.4356	0.7842	0.3530	0.7268
fasterrcnn_resnet50_fpn [2] w/ Fine-Tuning	0.4459 **(+1.03%)**	0.8098 **(+2.56%)**	0.3685 **(+1.55%)**	0.7660 **(+3.92%)**
fasterrcnn_resnet50_fpn_v2[3] o/ Fine-Tuning	0.4428	0.8172	0.3653	0.7896
fasterrcnn_resnet50_fpn_v2[3] w/ Fine-Tuning	–	–	–	–
fasterrcnn_resnet50_fpn_v2[3] + Residual FPN (ours) o/ Fine-Tuning	**0.4474**	**0.8186**	**0.3737**	**0.7916**
fasterrcnn_resnet50_fpn_v2[3] + Residual FPN w/ Fine-Tuning	**0.4496 (+0.22%)**	**0.8247 (+0.61%)**	**0.3777 (+0.40%)**	**0.7958 (+0.42%)**

Table 1. All experiments results

Based on the experimental results (note that due to time constraints, we were unable to complete all six model variants, and **therefore only selected the better-performing fasterrcnn_resnet50_fpn_v2[3] with Residual FPN (ours) for fine-tuning)**, we observed that both baseline models used in our experiments outperformed the strong baseline defined in this lab in terms of both mAP and accuracy.

Between the two baselines, fasterrcnn_resnet50_fpn_v2[3] exhibited superior performance on the test set, outperforming fasterrcnn_resnet50_fpn[2] by approximately **1.2% in mAP and 6.3% in accuracy without any fine-tuning.** However, **in contrast to the COCO-val2017 benchmark results—where fasterrcnn_resnet50_fpn_v2[3] reportedly improved upon its predecessor by 9.7% in mAP [3]—our results showed only marginal gains.**

We hypothesize that several factors may have limited the performance improvements observed with the v2 model in our setting:
(1) The dataset used in our lab **exhibits inconsistent image resolutions** and generally **smaller object sizes**, which may reduce the advantages offered by deeper or more semantically rich feature extractors.

(2) Due to hardware constraints, we used a **relatively small batch size of 2** for training fasterrcnn_resnet50_fpn_v2[3]. Given that this model **uses trainable Batch Normalization layers, the small batch size likely led to unstable batch statistics**, which in turn degraded model performance.

In addition to evaluating the two baseline models, we also conducted additional experiments to assess the effectiveness of our proposed method. Specifically, we introduced a modification to the fasterrcnn_resnet50_fpn_v2[3] architecture by appending a residual block after the original FPN [5] structure. Experimental results indicate that our method outperformed the original fasterrcnn_resnet50_fpn_v2[3] baseline on the test set, with an increase of approximately 0.84% in mAP and 0.2% in accuracy.

Although the performance gain is modest, it provides empirical support for the hypothesis we proposed in Section 2.2—namely, that introducing an additional residual block after multi-scale semantic fusion can enhance non-linear transformation capacity and deepen semantic representation, thereby offering more informative features to the detection head and contributing to overall model performance improvements.

In the fine-tuning phase, **we did not apply layer freezing or introduce additional architectural components.** Instead, we continued training from previously learned weights using a reduced learning rate. Additionally, we adopted a two-stage data augmentation strategy: during the first 8 epochs, data augmentation was enabled to enhance the model's generalization ability; in the subsequent 12 epochs, augmentation was disabled to provide more stable input, allowing the model to focus on learning finer-grained feature representations.

Under this setting, the fasterrcnn_resnet50_fpn[2] model achieved a 1.55% increase in mAP and a 3.92% improvement in accuracy on the test set after fine-tuning. Meanwhile, **our proposed model—fasterrcnn_resnet50_fpn_v2[3] equipped with the Residual FPN—achieved an additional 0.40% gain in mAP and 0.42% improvement in accuracy following fine-tuning.** Although the overall performance gains were modest, these results demonstrate that combining **a lower learning rate with a staged data augmentation schedule can effectively lead to incremental improvements in detection performance.**

4.2 Conclusion and Future Work

Our Experimental results indicate that the method implemented in our additional experiments—namely, **inserting a residual block after the FPN[5] structure**—can **effectively and consistently enhance the performance of the model on object detection tasks**, without significantly increasing the number of parameters or computational cost. However, in the current design, the enhancement was applied solely to the FPN module. Therefore, future work may consider more extensive modifications to these modules in order to achieve further improvements in overall model performance.

Regarding the Detection Head, beyond incorporating attention mechanisms to enhance semantic representations in key regions, adopting a multi-stage prediction strategy **such as a Cascade Head may also be beneficial**. By progressively refining classification and bounding box regression at each stage, this approach can improve localization accuracy and region discrimination, especially in high-IoU detection scenarios. As for the Neck architecture, **future directions could involve redesigning the anchor generation mechanism or incorporating spatially adaptive modules such as Deformable Convolutions.** These approaches may help improve the diversity and semantic consistency of region proposals.

Furthermore, the Backbone network itself presents another promising direction for enhancement. Recent research highlights the superior global representation capability of **Vision Transformers (ViTs) and their variants.** Incorporating ViTs into the Faster R-CNN framework—as a replacement for traditional CNN-based backbones—may improve the model's ability to understand complex object relationships, particularly in low-resolution or visually cluttered scenes.

Reference

[1] Cubuk, E. D., Zoph, B., Mane, D., Vasudevan, V., & Le, Q. V. (2018). Autoaugment: Learning augmentation policies from data. arXiv preprint arXiv:1805.09501.

[2]https://pytorch.org/vision/main/models/generated/torchvision.models.detection.fasterrcnn_resnet50_fpn.html

[3]https://pytorch.org/vision/master/models/generated/torchvision.models.detection.fasterrcnn_resnet50_fpn_v2.html

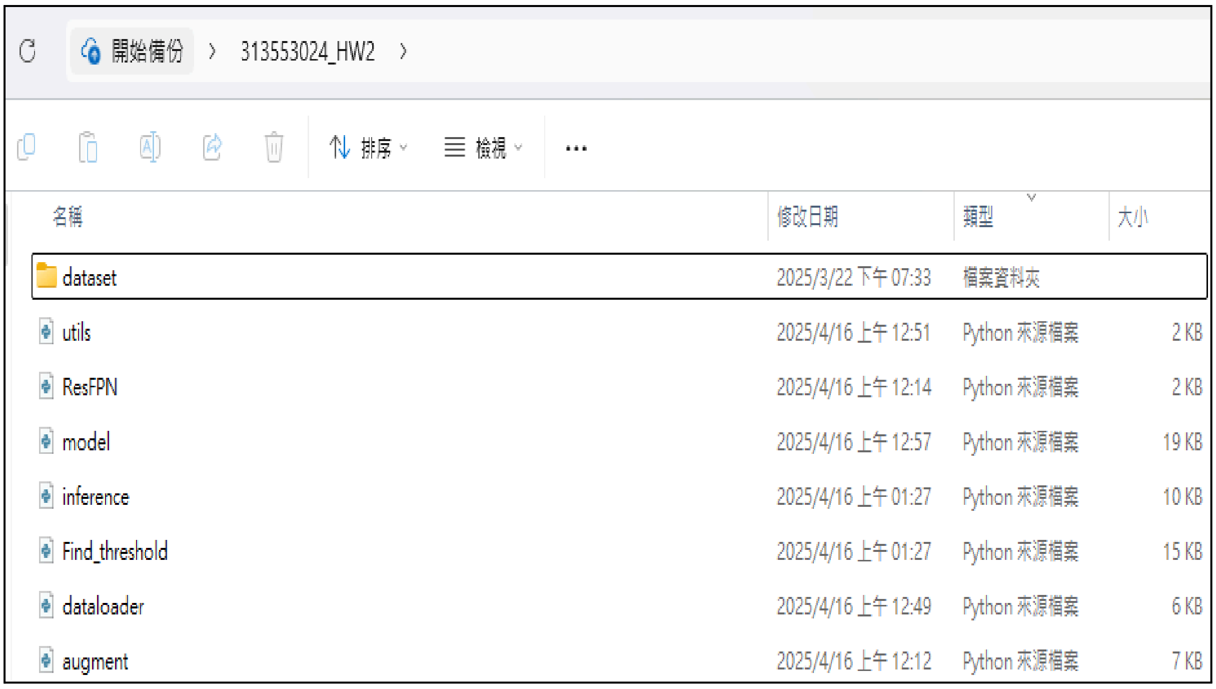
[4] Ge, Z., Liu, S., Wang, F., Li, Z., & Sun, J. (2021). Yolox: Exceeding yolo series in 2021. arXiv preprint arXiv:2107.08430.

[5] Lin, T. Y., Dollár, P., Girshick, R., He, K., Hariharan, B., & Belongie, S. (2017). Feature pyramid networks for object detection. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2117-2125).

Appendix

(1) File Structures:

The file structure for this experiment is depicted in the figure below. The downloaded dataset is placed in a folder named datasets, which contains several Python files. We use model.py for model training and, upon obtaining the optimal weight file, we utilize Find_threshold.py to determine the best threshold on the validation set; this threshold is subsequently applied during inference via inference.py. Additionally, the files used during the training process include **ResFPN.py (employed in supplementary experiments involving the Residual FPN)**, dataloader.py (responsible for loading the dataset), augment.py (which implements data augmentation techniques on the training set), and utils.py (which is used for plotting the training and validation loss curves, as well as the mAP curve on the validation set). All of these files and folders are packaged and stored in a directory named 313553024_HW2.



名稱	修改日期	類型	大小
dataset	2025/3/22 下午 07:33	檔案資料夾	
utils	2025/4/16 上午 12:51	Python 來源檔案	2 KB
ResFPN	2025/4/16 上午 12:14	Python 來源檔案	2 KB
model	2025/4/16 上午 12:57	Python 來源檔案	19 KB
inference	2025/4/16 上午 01:27	Python 來源檔案	10 KB
Find_threshold	2025/4/16 上午 01:27	Python 來源檔案	15 KB
dataloader	2025/4/16 上午 12:49	Python 來源檔案	6 KB
augment	2025/4/16 上午 12:12	Python 來源檔案	7 KB

Fig 18. Our File Structure with 7 python file and a datasets folder (which would be removed before we upload our .zip file to E3 and Github)

(2) Command Line in Inference phase:

a. Explanation

--use_res_fpn:
Enables the custom Residual FPN enhancement, replacing the default FPN with a stronger version. **If you do not want to use it, you do not need to type --use_re_fpn. Then you can use the default FPN.**

--train_root:
Specifies the root directory containing training images.

--train_ann:
Specifies the COCO-format JSON annotation file for the training set.

--valid_root:
Specifies the root directory containing validation images.

--valid_ann:
Specifies the COCO-format JSON annotation file for the validation set.

--train_bs:
Sets the batch size for training.

--valid_bs:
Sets the batch size for validation.

--num_epochs:
Defines the total number of epochs to train the model.

--warmup_epochs:
Specifies the number of warmup epochs during which the learning rate gradually increases.

--lr:
Sets the initial learning rate for training.

--eta_min:
Defines the minimum learning rate for the cosine annealing scheduler.

--num_workers:
Specifies the number of worker processes to use for data loading.

--batch_size (in Find_threshold.py and inference.py):
Sets the batch size for processing images during threshold finding or inference.

--checkpoint:
Specifies the model checkpoint file to load during the inference phase.

--finetune_weights:
Specifies a pre-trained weight file to load for fine-tuning instead of starting from scratch.

b. **Training command line:**

```
python model.py --use_res_fpn --train_root dataset/train --train_ann dataset/train.json
--valid_root dataset/valid --valid_ann dataset/valid.json --train_bs 2 --valid_bs 4
--num_epochs 30 --warmup_epochs 5 --lr 1e-4 --eta_min 5e-6 --num_workers 0
```

c. **Find the threshold we will use in inference phase:**

```
python Find_threshold.py --use_res_fpn --train_root dataset/train --train_ann
dataset/train.json --valid_root dataset/valid --valid_ann dataset/valid.json --batch_size
4
```

d. **Inference command line:**

```
python inference.py --use_res_fpn --batch_size 8 --test_root dataset/test --checkpoint
best_resfpn_v2.pth
```

e. **Fine-Tuning Command line:**

```
python model.py --use_res_fpn --train_root dataset/train --train_ann dataset/train.json
--valid_root dataset/valid --valid_ann dataset/valid.json --train_bs 2 --valid_bs 4
--num_epochs 20 --warmup_epochs 1 --lr 5e-6 --eta_min 5e-8 --num_workers 0
--finetune_weights best.pth
```

(3) Github Link:
https://github.com/rmd926/NYCU_CV2025_Spring/tree/main/Lab2