Report

110550128 蔡耀霆

Github: https://github.com/WhiteOuO/VRDL_HW2

Introduction:

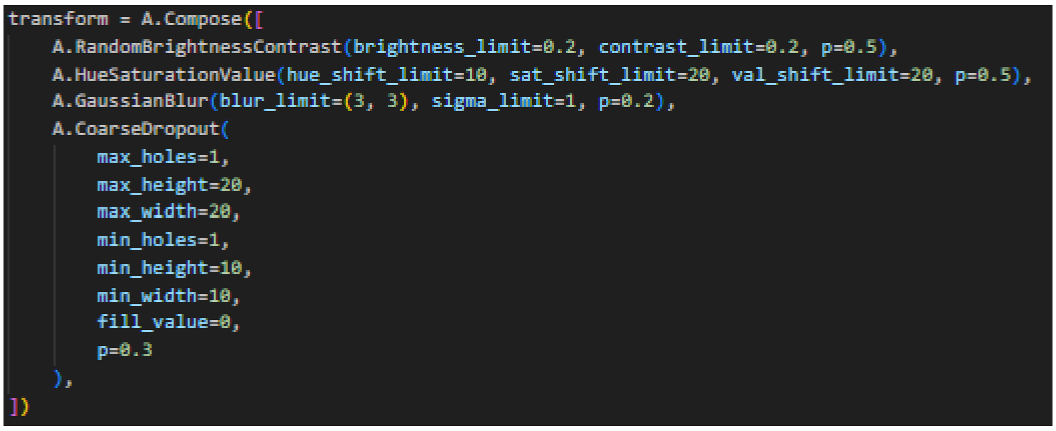
In this project, we aim to solve a visual recognition task that involves both object detection and digit recognition.

- **Task 1:** Detect all digits in the image by predicting their bounding boxes and class labels.
- **Task 2:** Sort the predicted digits by their x-coordinates and concatenate them to form the final number.

The model we use in this task is more complex than the one used previously. It consists of a backbone network, a Region Proposal Network (RPN), RoI Pooling, and a detection head. Task 1 heavily relies on the RPN, which functions as a region-based classifier responsible for identifying potential regions in the image that may contain digits. These proposed regions are then refined by the RoI Pooling and the detection head to generate the final bounding boxes.

Method:

Datasets Preprocess:



```
transform = A.Compose([
    A.RandomBrightnessContrast(brightness_limit=0.2, contrast_limit=0.2, p=0.5),
    A.HueSaturationValue(hue_shift_limit=10, sat_shift_limit=20, val_shift_limit=20, p=0.5),
    A.GaussianBlur(blur_limit=(3, 3), sigma_limit=1, p=0.2),
    A.CoarseDropout(
        max_holes=1,
        max_height=20,
        max_width=20,
        min_holes=1,
        min_height=10,
        min_width=10,
        fill_value=0,
        p=0.3
    ),
])
```

I applied the above techniques to perform data augmentation on my dataset. Afterward, I converted the augmented images into .pt files. This allows me to avoid spending computational resources and time on augmentation during data loading. However, this approach also led to memory resource limitations later on.

Computational resources limitation and Solution:

During the implementation of this object detection task using the Faster R-CNN model, I encountered significant computational and memory limitations. The dataset was large and required extensive training, which placed a heavy load on RAM. In my design, all training data was preprocessed into .pt tensors(I expected to lessen the training workload. Do the data augmentation and turn into tensors in another file beforehand.) and loaded entirely into memory using a preloading strategy.(Reduce the disk I/O times, instead of lazy loader) When both training and validation sets were loaded simultaneously, it caused memory thrashing, where the system constantly swapped memory pages between RAM and disk, severely slowing down training. As a result, a single epoch could take up to 2–3 hours.

To address this issue, I implemented a partitioned training strategy. I randomly split the full training set into 10 folders. During each training cycle, only one of the ten subsets was loaded into memory. This reduced memory usage significantly and eliminated thrashing, resulting in much faster training while maintaining data diversity across epochs.

In addition to memory optimization, I conducted the following three experiments:

1. Dropout Regularization: I added a dropout layer after the fully connected layer in the detection head of the model to mitigate overfitting.

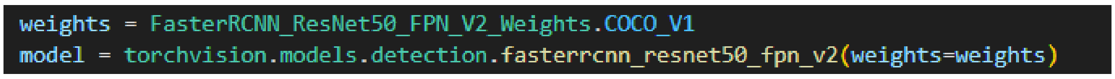
2. **Test-Time Super Resolution**: To enhance digit clarity in test images, I applied 2× upscaling using Lanczos interpolation. To ensure that the bounding box coordinates remained aligned with the original image scale, all predicted bounding boxes were divided by 2 after inference.

3. **Validation Removal**: I decided to merge the validation data into the training set in order to increase the total amount of data the model can

learn from.

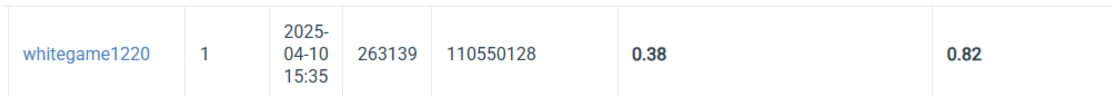
These modifications allowed the model to be trained effectively under constrained hardware conditions and improved both training stability and efficiency.

The pre-trained model and weights I used:



```
weights = FasterRCNN_ResNet50_FPN_V2_Weights.COCO_V1
model = torchvision.models.detection.fasterrcnn_resnet50_fpn_v2(weights=weights)
```

Results & Findinds:



whitegame1220	1	2025-04-10 15:35	263139	110550128	**0.38**	**0.82**

Why I’m not using validation?

The Faster R-CNN model with a ResNet-50 backbone is able to finish training on all 30,000 training images within a little over an hour. One key reason for this efficiency is that the model does not perform full validation after every epoch (here, one “epoch” means a pass through 1/10 of the training set). This design significantly reduces memory pressure, as the validation data does not permanently occupy RAM.

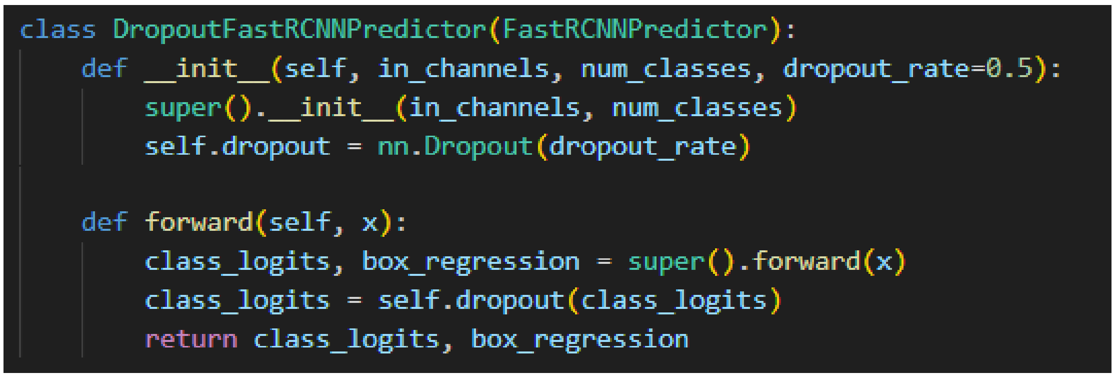
Instead of maintaining a dedicated validation set, I divide the entire training data into 10 subsets. At each epoch, only one subset is loaded into memory. During the first 10 epochs, the model first computes the loss on a batch before performing any weight updates. This means the loss reflects the model’s performance on previously unseen data at that moment, making it a reliable proxy for validation loss. These losses act like validation loss, reflecting how the model generalizes to previously unseen data, even though they are technically part of the training data.

This design offers another benefit: fast and adaptive learning rate scheduling. Since the model is exposed to new data at every epoch, we can dynamically adjust the learning rate to accelerate convergence and reduce oscillations near local minimum. I also designed a decay mechanism that prevents learning rate reduction before the model has seen all subsets at least once, which leads to potential imbalance where later data would have a smaller effect on the model.

Additionally, since no validation set is needed, I incorporated the validation data into the training set. This gives the model access to more data, which can further improve generalization.

Compared to methods like lazy loading—where each batch is read directly from disk on demand—The method I proposed avoids disk I/O bottlenecks problem from lazy loading while maintaining low memory usage. It strikes a balance between efficiency and effectiveness, and thus I adopted it for all my experiments. Furthermore, I believe that we use different data as validation set will be a more proper way to approach ground truth.

Dropout:

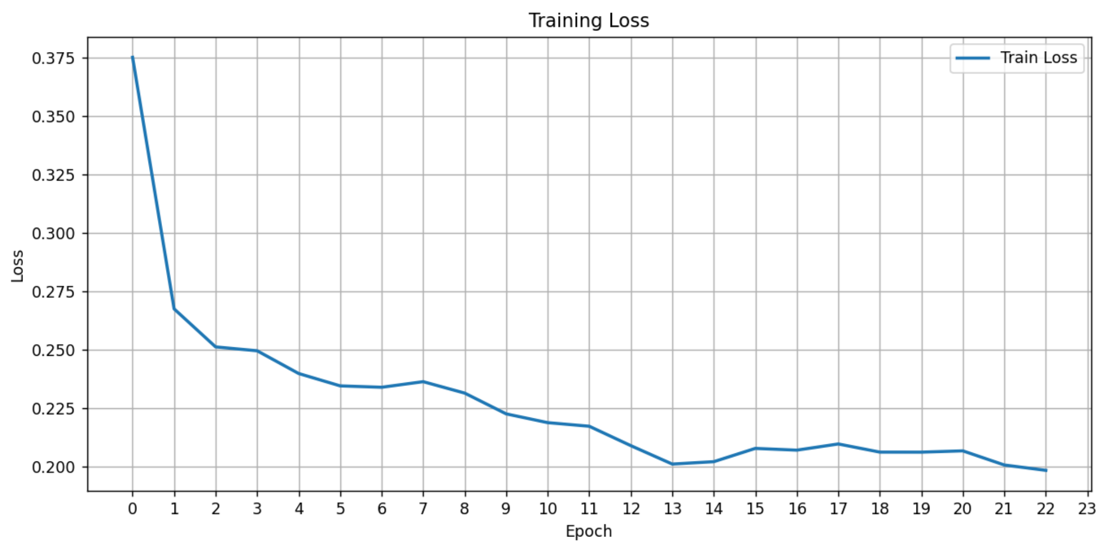


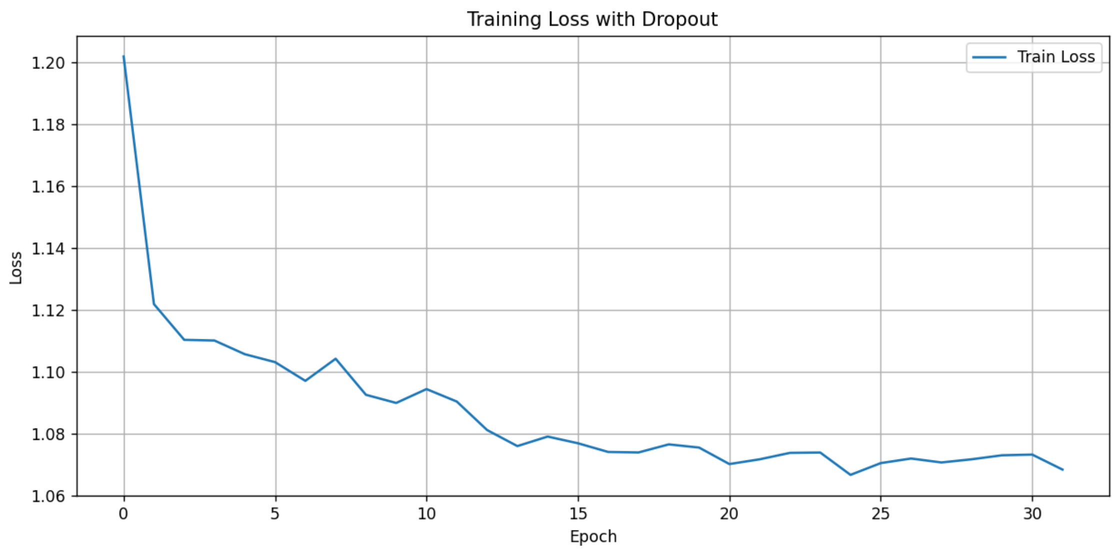
```
class DropoutFastRCNNPredictor(FastRCNNPredictor):
    def __init__(self, in_channels, num_classes, dropout_rate=0.5):
        super().__init__(in_channels, num_classes)
        self.dropout = nn.Dropout(dropout_rate)

    def forward(self, x):
        class_logits, box_regression = super().forward(x)
        class_logits = self.dropout(class_logits)
        return class_logits, box_regression
```

To reduce overfitting and improve generalization, we modified the classification head of the Faster R-CNN model by adding a Dropout layer. We created a custom predictor class that inherits from FastRCNNPredictor and inserts a Dropout layer after the classification logits are computed. This layer randomly drops some neuron outputs during training, which helps prevent the model from relying too heavily on specific features. Importantly, we only apply Dropout to the classification branch, not the bounding box regression branch, to maintain the stability of bounding box predictions.

We will use the prediction on test data and training(validation) loss line chart to evaluate the efficiency of dropout layer.

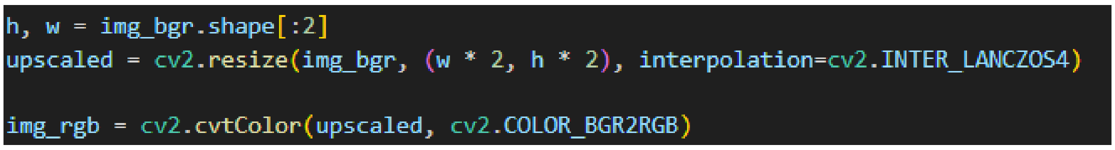




In this experiment, adding a Dropout layer did not prove to be an effective method for improvement. The model's loss remained significantly high — even after 30 epochs of training, the loss did not fall below three times the value of the first epoch's loss of the version without Dropout.

While Dropout is effective in fully connected layers to reduce overfitting, it may not be well suited for object detection tasks such as Faster R-CNN. In this case, Dropout was added after the detection head, which might interfere with the extraction of critical spatial features. Since object detection relies heavily on precise spatial information, the random suppression of neuron outputs caused by Dropout can hinder the model's ability to consistently focus on relevant regions during training. This likely explains why the performance degraded instead of improving.

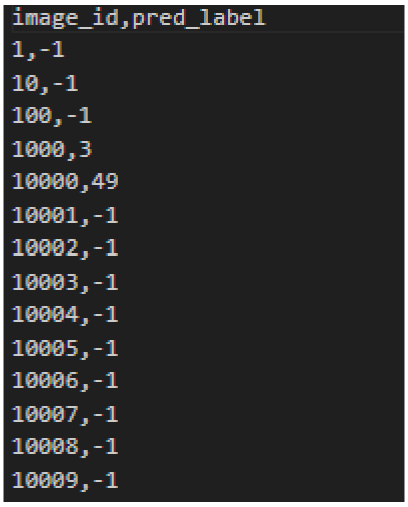
Test super resolution:



```
h, w = img_bgr.shape[:2]
upscaled = cv2.resize(img_bgr, (w * 2, h * 2), interpolation=cv2.INTER_LANCZOS4)

img_rgb = cv2.cvtColor(upscaled, cv2.COLOR_BGR2RGB)
```

I decided to conduct this experiment based on my observations that the model performs poorly on low-resolution images. Therefore, I plan to transform my test data into a higher-resolution version. Specifically, I use an interpolation method to upscale each image by a factor of two in both width and height. However, it is important to adjust the predicted bounding boxes by dividing their coordinates by 2, in order to map them back to the original image scale.



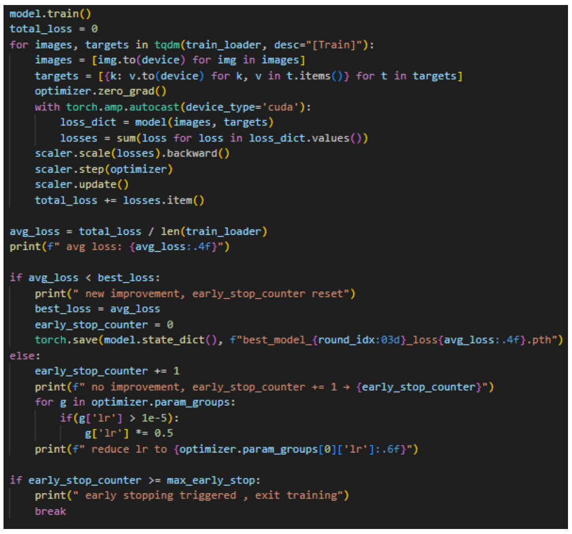
```
image_id,pred_label
1,-1
10,-1
100,-1
1000,3
10000,49
10001,-1
10002,-1
10003,-1
10004,-1
10005,-1
10006,-1
10007,-1
10008,-1
10009,-1
```

Based on the results, only a few instances were successfully detected. After reflecting on the issue, I realized the problem: I should have also applied super resolution to the training data. In addition, adjusting the anchor size parameter—which defines the initial proposal areas in the model—would help the model better adapt to the scale of bounding boxes in both the training and testing data. The original model may have learned to generate bounding boxes of certain sizes from the training data, but on the super-resolved test data, these boxes might only cover part of a digit, which significantly weakens the model's ability to recognize complete digits.

Train and Valid:

Finally, I still want to do an experiments on validation. I believe this approach has several advantages. First, using different validation sets in each round of training provides more diverse evaluation and may be more effective than using a fixed validation set. Second, by incorporating the original validation set into training, the model can learn from more data, potentially improving its generalization ability. In addition, training time is shorter and memory usage is lower. We will compare the performance of the model trained with this method.

Validate while training version



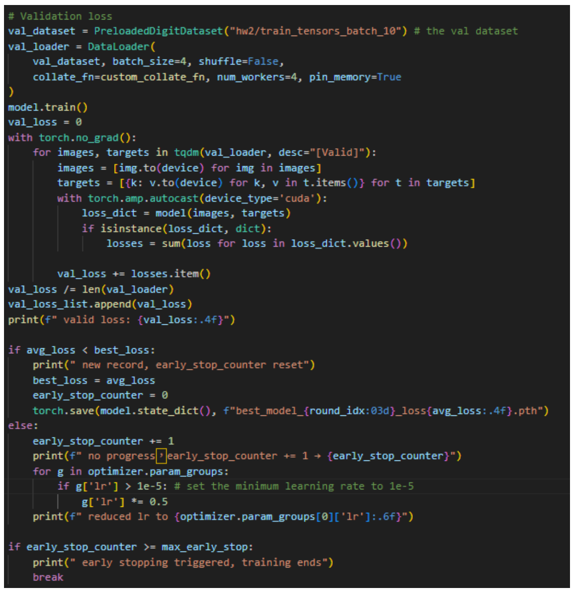
```python
model.train()
total_loss = 0
for images, targets in tqdm(train_loader, desc="[Train]"):
    images = [img.to(device) for img in images]
    targets = [{k: v.to(device) for k, v in t.items()} for t in targets]
    optimizer.zero_grad()
    with torch.amp.autocast(device_type='cuda'):
        loss_dict = model(images, targets)
        losses = sum(loss for loss in loss_dict.values())
    scaler.scale(losses).backward()
    scaler.step(optimizer)
    scaler.update()
    total_loss += losses.item()

avg_loss = total_loss / len(train_loader)
print(f" avg loss: {avg_loss:.4f}")

if avg_loss < best_loss:
    print(" new improvement, early_stop_counter reset")
    best_loss = avg_loss
    early_stop_counter = 0
    torch.save(model.state_dict(), f"best_model_{round_idx:03d}_loss{avg_loss:.4f}.pth")
else:
    early_stop_counter += 1
    print(f" no improvement, early_stop_counter += 1 → {early_stop_counter}")
    for g in optimizer.param_groups:
        if(g['lr'] > 1e-5):
            g['lr'] *= 0.5
    print(f" reduce lr to {optimizer.param_groups[0]['lr']:.6f}")

if early_stop_counter >= max_early_stop:
    print(" early stopping triggered , exit training")
    break
```

Fixed Validation set version

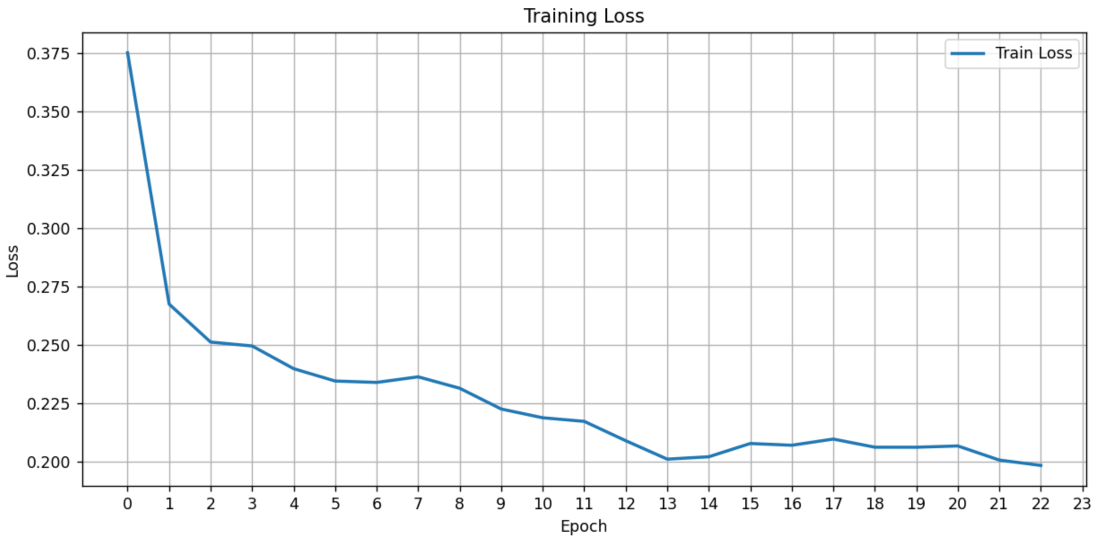


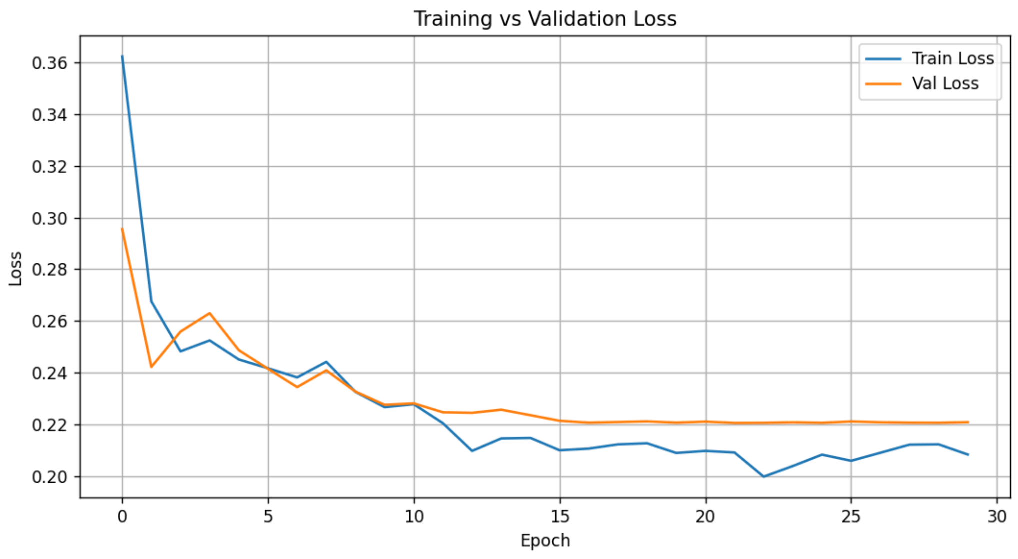
```python
# Validation loss
val_dataset = PreloadedDigitDataset("hw2/train_tensors_batch_10") # the val dataset
val_loader = DataLoader(
    val_dataset, batch_size=4, shuffle=False,
    collate_fn=custom_collate_fn, num_workers=4, pin_memory=True
)
model.train()
val_loss = 0
with torch.no_grad():
    for images, targets in tqdm(val_loader, desc="[Valid]"):
        images = [img.to(device) for img in images]
        targets = [{k: v.to(device) for k, v in t.items()} for t in targets]
        with torch.amp.autocast(device_type='cuda'):
            loss_dict = model(images, targets)
            if isinstance(loss_dict, dict):
                losses = sum(loss for loss in loss_dict.values())

        val_loss += losses.item()
val_loss /= len(val_loader)
val_loss_list.append(val_loss)
print(f" valid loss: {val_loss:.4f}")

if avg_loss < best_loss:
    print(" new record, early_stop_counter reset")
    best_loss = avg_loss
    early_stop_counter = 0
    torch.save(model.state_dict(), f"best_model_{round_idx:03d}_loss{avg_loss:.4f}.pth")
else:
    early_stop_counter += 1
    print(f" no progress，early_stop_counter += 1 → {early_stop_counter}")
    for g in optimizer.param_groups:
        if g['lr'] > 1e-5: # set the minimum learning rate to 1e-5
            g['lr'] *= 0.5
    print(f" reduced lr to {optimizer.param_groups[0]['lr']:.6f}")

if early_stop_counter >= max_early_stop:
    print(" early stopping triggered, training ends")
    break
```





The final predictions on the test set are almost the same for both versions. Although the "validate while training" version performs slightly better, I believe the difference is within the margin of error. However, in terms of training time, I think this modification is well worth it.