Digit Recognition using Faster R-CNN

Chu-Chun, Chi

April 14, 2025

1 Introduction

https://github.com/chuchunchi/HW2_FasterRCNN_DigitRECOG

This project implemented a digit recognition task using a modified Faster R-CNN[1] architecture. The objective was to 1. Find the label and bounding box of each digit on a RGB image and 2. Predict the whole number in the image. What made my work different is the implementation of the Multi-Orientation Digit Recognition Algorithm to deal with task 2 (detailed in Section 2.3).

The core idea of my method includes:

- A fasterRCNN model with backbone ResNet 50[2]
- A final classifier with output dimension = 11
- A multi-orientation digit recognition algorithm to filter the result of task 1
- Tuning on different confident threshold and optimizer

2 Method

This section details our approach to the digit recognition task using Faster R-CNN. We describe the data processing pipeline, the model architecture with specific focus on the backbone, neck (RPN), and head components, and the hyperparameters used for training.

2.1 Data Processing

2.1.1 Dataset Handling

The dataset consists of RGB images containing digits that need to be detected and recognized. To efficiently handle the dataset, we implemented a custom `DigitDataset` class that inherits from PyTorch's `Dataset`[3] class. The dataset includes training, validation, and test splits with 30,062, 3,340, and 13,068 images respectively.

We initially approached data loading by preloading all images into memory:

```python
self.images = {}
for image in self.img_files:
    img_path = os.path.join(img_dir, image)
    img = Image.open(img_path).convert('RGB')
    self.images[image] = img.copy()
```

However, this approach led to memory exhaustion issues due to the large dataset size. We resolved this by loading images on-demand in the `__getitem__` method:

```python
def __getitem__(self, idx):
    img_file = self.img_files[idx]
    img_path = os.path.join(self.img_dir, img_file)
    img = Image.open(img_path).convert('RGB')
```

During implementation, we encountered an ”open too many files” OS error because file handles were not being properly closed. We resolved this issue by using the context manager pattern (`with` statement) to ensure proper resource cleanup:

```python
with open(img_path, 'rb') as f:
    img = Image.open(f).convert('RGB')
```

2.1.2 COCO Annotation Processing

The dataset annotations are provided in COCO format, which required specialized handling. We utilized the `pycocotools`[4] library to parse and process these annotations. To optimize performance, we preprocessed the annotations during initialization:

```python
def _preprocess_annotations(self):
    for img_id in self.img_ids:
        ann_ids = self.coco.getAnnIds(imgIds=img_id)
        if len(ann_ids) > 0:
            anns = self.coco.loadAnns(ann_ids)

            # Extract boxes, labels, and areas
            boxes = []
            labels = []
            areas = []

            for ann in anns:
                # COCO bbox format: [x_min, y_min, width,
                    height]
                x, y, w, h = ann['bbox']

                # Convert to [x_min, y_min, x_max, y_max]
                    format
                boxes.append([x, y, x + w, y + h])

                # Convert category_id to 0-9 (COCO starts
                    from 1)
                labels.append(ann['category_id'] - 1)  # 
                    Convert to 0-indexed
                areas.append(w * h)

            self.annotations[img_id] = {
                'boxes': torch.as_tensor(boxes, dtype=torch.
                    float32),
                'labels': torch.as_tensor(labels, dtype=
                    torch.int64),
                'areas': torch.as_tensor(areas, dtype=torch.
                    float32),
                'iscrowd': torch.zeros(len(boxes), dtype=
                    torch.int64)
            }
```

A key aspect of the preprocessing was the conversion of bounding box coordinates from COCO format [x_min, y_min, width, height] to the [x_min, y_min, x_max, y_max] format required by the Faster R-CNN implementation. Additionally, we adjusted the category IDs by subtracting 1 to convert from COCO's 1-indexed to our 0-indexed labeling system.

2.1.3 Data Augmentation

To improve model generalization, we implemented data augmentation techniques for the training dataset[5]:

```
if train:
    transforms = [
        T.RandomRotation(10),
        T.ColorJitter(brightness=0.2, contrast=0.2,
            saturation=0.2),
    ] + transforms
```

The augmentation pipeline includes:

- **Random rotation**: Images are randomly rotated by up to 10 degrees to make the model robust to digit orientation variations.
- **Color jittering**: Brightness, contrast, and saturation are randomly adjusted within a range of $\pm 20\%$ to help the model become invariant to lighting conditions.

For both training and evaluation, we applied standard normalization using ImageNet mean and standard deviation values:

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

2.2 Model Architecture

Our solution is based on the Faster R-CNN architecture, which consists of three main components: backbone, neck (Region Proposal Network), and head. We implemented this through a custom `DigitDetector` class that encapsulates both Task 1 (digit detection) and Task 2 (whole number recognition).

2.2.1 Backbone

For the backbone network, we chose ResNet-50 with Feature Pyramid Network (FPN) as the feature extractor. ResNet-50 is a deep convolutional neural network with residual connections that help mitigate the vanishing gradient problem, allowing for effective training of deeper networks. The model was initialized with weights pre-trained on ImageNet, leveraging transfer learning to improve performance on our specific task:

```
    model = torchvision.models.detection.
        fasterrcnn_resnet50_fpn(
        pretrained=cfg.pretrained,
        pretrained_backbone=cfg.pretrained
    )
```

The Feature Pyramid Network (FPN) enhances the backbone by creating a multi-scale feature pyramid, which significantly improves the detection of objects (digits) at different scales. This is particularly important for our task since digits can appear at various sizes within images.

2.2.2 Neck (Region Proposal Network)

The Region Proposal Network (RPN) serves as the ”neck” of our Faster R-CNN architecture. It generates region proposals that potentially contain objects of interest, in our case, digits. We use default anchor configuration from torchvision.

The RPN was configured in the model as follows:

```
model.rpn.nms_thresh = 0.7
model.rpn.fg_iou_thresh = 0.7
model.rpn.bg_iou_thresh = 0.3
```

2.2.3 Head

The head of our Faster R-CNN model consists of two branches: a classification branch to identify the digit class (0-9 or background) and a regression branch to refine the bounding box coordinates. We replaced the default classifier with a custom one tailored to our 11-class problem:

```
in_features = model.roi_heads.box_predictor.cls_score.
    in_features
model.roi_heads.box_predictor = FastRCNNPredictor(
    in_features,
    cfg.num_classes + 1  # +1 for background class
)
```

The head parameters were configured with the following settings:

- **Score threshold**: 0.05
- **NMS threshold**: 0.5
- **Maximum detections per image**: 100

These settings were chosen to balance between recall and precision in the detection task. The relatively low score threshold ensures we don't miss potential digits, while the NMS threshold helps eliminate redundant detections.

2.3 Multi-Orientation Digit Recognition Algorithm

2.3.1 Problem Statement

In the digit recognition task, after detecting individual digits using Faster R-CNN, we need to correctly combine these digits to form the complete number. However, this presents several challenges:

- Numbers can appear in various orientations: horizontal, vertical, or with a slope
- Multiple numbers may appear in a single image
- Some digits may have low confidence scores or false positives

Our initial approach focused on horizontal alignment, filtering digits based on their vertical positions. While this worked for standard horizontally-aligned numbers, it failed for other orientations, such as vertically stacked digits.

2.3.2 Proposed Algorithm

To address these challenges, we developed an adaptive clustering algorithm that can handle digits in any orientation. The algorithm consists of the following steps:

1. **Confidence Filtering:** Filter detected digits by a confidence threshold (optimally 0.7).
2. **Orientation Analysis:** Determine whether digits are arranged horizontally or vertically by comparing the spans of x and y coordinates.
3. **Adaptive Clustering:** Group digits that belong to the same number using a spatial clustering approach. The clustering uses a dynamic spacing threshold based on the average digit size.

4. **Orientation-Based Sorting:** Sort digits within each cluster according to the detected orientation:
    - Horizontal arrangement: Sort from left to right
    - Vertical arrangement: Sort from top to bottom
5. **Number Formation:** Combine the sorted digits to form the complete number.

2.3.3 Implementation Details

The core of our approach is the adaptive clustering algorithm. Instead of using a fixed alignment threshold, we compute the spacing dynamically based on the average digit size:

```python
# Calculate box dimensions
widths = boxes[:, 2] - boxes[:, 0]
heights = boxes[:, 3] - boxes[:, 1]
avg_width = np.mean(widths)
avg_height = np.mean(heights)

# Calculate spacing threshold based on average size
spacing_threshold = max(avg_width, avg_height) * 1.5
```

Listing 1: Adaptive spacing threshold calculation

The clustering algorithm groups digits that are spatially close to each other, regardless of their specific arrangement:

```python
# Cluster digits that belong to the same number
clusters = []
visited = set()

for i in range(len(boxes)):
    if i in visited:
        continue

    cluster = [i]
    visited.add(i)

    # Find all connected digits
    queue = [i]
    while queue:
        current = queue.pop(0)

        for j in range(len(boxes)):
            if j in visited:
                continue

            # Calculate distance between centers
            dist = np.linalg.norm(centers[current] - centers
                [j])

            if dist < spacing_threshold:
                cluster.append(j)
                visited.add(j)
                queue.append(j)

    clusters.append(cluster)
```

Listing 2: Clustering algorithm for digit grouping

3 Experiments and Results

3.1 Experiments

The model was trained on Geforce RTX 4090 GPU and have been trained for 24 epochs, with the following hyperparameters:

- **Batch size**: 8
- **Number of workers**: 4
- **Learning rate**: 0.001
- **Weight decay**: 0.0001
- **Number of epochs**: 24
- **Optimizer**: AdamW
- **Learning rate scheduler**: OneCycleLR with cosine annealing strategy (10% warm-up period)

We initially experimented with SGD with momentum but found that the AdamW optimizer combined with the OneCycleLR scheduler yielded better performance. AdamW applies weight decay correctly, unlike traditional Adam, and the OneCycleLR scheduler provides a more sophisticated learning rate policy with warm-up and cosine annealing. The result shows that given the same training epochs, AdamW optimizer combined with the OneCycleLR scheduler perform better, probably because the adaptive learning rate scheduler and the proper weight decay algorithm AdamW implemented.

```
optimizer = torch.optim.AdamW(
    params,
    lr=cfg.learning_rate,
    weight_decay=cfg.weight_decay  # AdamW applies proper
        weight decay
)

lr_scheduler = torch.optim.lr_scheduler.OneCycleLR(
    optimizer,
    max_lr=cfg.learning_rate,
    total_steps=cfg.num_epochs * len(train_loader) + 5,  #
        extra buffer
    pct_start=0.1,  # Spend 10% of training time warming up
    anneal_strategy='cos',
    div_factor=25.0   # Initial LR is max_lr/25
)
```

During training, we monitored the validation loss and saved model checkpoints, with special emphasis on preserving the best-performing model (model with lowest validation loss):

```
if val_loss < best_loss:
    best_loss = val_loss
    best_path = os.path.join(cfg.output_dir, 'best_model.pth
        ')
    torch.save(checkpoint, best_path)
```

In addition, we have tried different 'task1-to-task2-generation' approach, and we finally adapt the multi-orientation digit grouping algorithm with fine-tuning confident threshold=0.7 that yields the best task 2 accuracy, detailed experiments result will be shown below.

3.2 Result

3.2.1 Multi-Orientation Algorithm Results

We experimented with different approaches for digit combination and evaluated their performance in the validation set with the same best_model.pth, please find the visualization result in Figure 1.

Method	Accuracy
Baseline (no y-alignment, threshold=0.6)	0.6832
Simple y-alignment, threshold=0.6	0.6715
Adaptive clustering, threshold=0.5	0.6596
Adaptive clustering, threshold=0.6	0.6850
Adaptive clustering, threshold=0.7 (best)	**0.6919**

Table 1: Performance comparison of different digit combination approaches

3.2.2 Discussion

The adaptive clustering approach outperforms the simple y-alignment filtering, achieving the best accuracy of 69.19% with a confidence threshold of 0.7. This improvement is mainly attributed to:

- **Orientation Flexibility:** The algorithm can handle numbers in any orientation, whether horizontal, vertical, or sloped.
- **Adaptive Spacing:** The dynamic spacing threshold adjusts to different digit sizes and spacings.

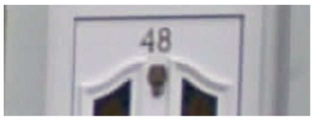
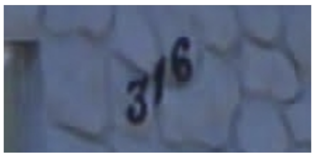
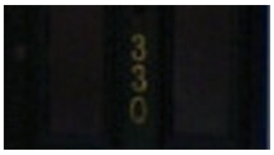
Example Image	Orientation	Baseline	Only Y-align	Our Method
	horizontal	498	-1	48
	slope	316	1	316
	vertical	033	-1	330

Figure 1: Comparison of digit recognition methods across different orientations. The baseline method fails on non-horizontal arrangements, the Y-alignment method fails on these cases because it want to align the output digits' y coordinates, while our proposed method correctly handles all orientations.

- **Robustness to Multiple Numbers:** The clustering approach can separate multiple numbers in the same image.

A higher confidence threshold (0.7) helped filter out false positives, which was more effective than relying on geometric constraints alone. The trade-off is that very low-confidence true positives might be missed, but the overall accuracy improved.

3.2.3 Final Model Result

The final model demonstrating strong performance across the 100 categories, specifically:

- Train Loss: 0.1533
- Validation Loss: 0.2335
- Validation Accuracy (Task 2): 0.729
- Testing accuracy: 0.71 (which beats the public strong baseline)

Table 2 presents the evaluation results of our model using COCO metrics. The model achieves an AP@0.5 (Average Precision at IoU threshold of 0.5) of 0.840, indicating strong performance for the basic detection task. The mean Average

Table 2: COCO Evaluation Metrics for Digit Detection

Average Precision (AP)		
IoU	**Area**	**Value**
0.50:0.95	all	0.409
0.50	all	**0.840**
0.75	all	0.330
0.50:0.95	small	0.399
0.50:0.95	medium	0.448
0.50:0.95	large	0.617

Average Recall (AR)		
IoU	**Area/MaxDets**	**Value**
0.50:0.95	all (maxDets=1)	0.458
0.50:0.95	all (maxDets=10)	0.494
0.50:0.95	all (maxDets=100)	0.494
0.50:0.95	small	0.487
0.50:0.95	medium	0.520
0.50:0.95	large	0.623

Precision (mAP) across IoU thresholds from 0.5 to 0.95 is 0.409, which reflects the model's ability to produce accurate bounding boxes. Notably, the model performs better on larger digits (AP=0.617) compared to small (AP=0.399) and medium-sized (AP=0.448) digits, suggesting that scale remains a challenging factor. The Average Recall metrics follow a similar pattern, with higher performance on larger digits. These results demonstrate that our Faster R-CNN implementation effectively detects digits across various scales, with particularly strong performance at the standard IoU threshold of 0.5 used in many object detection benchmarks.

4 Reference

1. Ren, S., He, K., Girshick, R., Sun, J. (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. Advances in Neural Information Processing Systems, 28.

2. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

3. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems, 32.

4. pycocotools: https://pypi.org/project/pycocotools/

5. transforms: https://pytorch.org/vision/0.9/transforms.html