Report on Comparative Analysis of Multiple Models for Blood Cell Detection Dataset

Ye Wang

April 2025

1 Introduction

This report uses different mainstream visual detection pre-trained models for the blood cell detection dataset, fine-tunes the dataset to achieve the correct recognition standard, and analyzes the differences between different models.

This article will introduce the work I did based on my notebook in the process of dataset, data flow, model, benchmark, code walk thought and result.

The notebook begins with data pre-processing steps, including image loading, label association, and visualization. Several deep learning models are then trained and assessed to determine their suitability for this task. This comparative study highlights the strengths and weaknesses of each model.

2 Dataset

This project uses a blood cell detection dataset.*This dataset contains annotated red blood cells(RBC) and white blood cells(WBC) from peripheral blood smear taken from a light microscope.*The dataset contains 100 annotated images. Each image has RGB three channels, 256 pixels in height and width, and is stored in PNG format. The annotations are stored in CSV files.

It contains 100 annotated images with labeled **RBC as 2237and WBC as 103**.I noticed this early in the training, the dataset has a large difference in the number of labels in the two categories, which may lead to low recognition rate for the minority label.But as the training progressed, I found that the characteristics of the two types of cells were obviously different. After sufficient training, the recognition accuracy of both types of cells was very high.

3 Data flow

The data processing pipeline of this project consists of the following key steps:

1. **Data Loading and Visualization**
   The annotation file (`annotations.csv`) is read to obtain image names and corresponding bounding box information. A subset of images is visualized with annotated boxes to verify the labeling accuracy and format.

2. **Dataset Splitting**
   The dataset is randomly divided based on image names into a training set (80%) and a validation set (20%). This ensures that all models are trained and evaluated on the same data partition for fair comparison.

3. **Format Conversion**
   Two target detection formats are prepared:

   - **YOLO format**: Each image's annotations are converted into the format `[class_id, x_center, y_center, width, height]` and saved as `.txt` files.
   - **COCO format**: Annotation data is structured into a JSON file containing image metadata, object categories, and bounding boxes, suitable for models such as Faster R-CNN and SSD.

4. **Data Loading for Training**
   According to the data in different formats, a custom data set class is formed to implement the data interface for model training and verification. And training is performed on the corresponding model

5. **Model Inference**
   Model performance is assessed through inference speed and detection accuracy, with visualizations of predicted bounding boxes on sample validation images.

4 Model Training and Evaluation

4.1 Model Overview

To evaluate the effectiveness of different object detection approaches on the blood cell dataset, three representative models were selected:

- **YOLOv8**: A state-of-the-art real-time object detector optimized for speed and accuracy.
- **Faster R-CNN**: A two-stage detector known for high accuracy, commonly used in medical imaging.
- **SSD300 (VGG16)**: A single-shot detector that balances detection performance and computational efficiency.

4.2 Training Strategy

All models are fine-tuned using the same training and validation sets to ensure fair comparison. The training processes follow the respective libraries' standard procedures:

- YOLOv8 is trained using Ultralytics with a configuration YAML file, 256 image size, and 50 epochs.
- Faster R-CNN is trained using the PyTorch `torchvision` implementation, with 100 epochs and SGD optimizer.
- SSD300 is trained with the VGG16 backbone, using a batch-based PyTorch loop for 100 epochs.

Each model's predicted outputs are converted to COCO format for evaluation using the `pycocotools` library.

5 Benchmark and Performance Comparison

To evaluate the effectiveness of the selected models, we conducted a comprehensive benchmark on the validation set using the COCO evaluation metrics. The following aspects were considered:

5.1 Evaluation Metrics

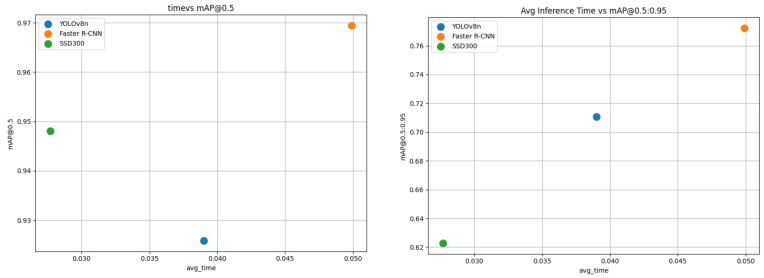
- **mAP@0.5**: Mean Average Precision at an IoU threshold of 0.5, indicating how well the model detects objects with acceptable overlap.
- **mAP@0.5:0.95**: Averaged mAP over IoU thresholds from 0.5 to 0.95 (in steps of 0.05), offering a more stringent and balanced evaluation.
- **mAP@0.75**: Precision at a higher IoU threshold, reflecting stricter localization accuracy.
- **Inference Time**: Average prediction time per image, used to measure model efficiency and suitability for real-time scenarios.

5.2 Results

The performance of each model is summarized below:

Model	mAP@0.5	mAP@0.75	mAP@0.5:0.95	Avg. Inference Time (s)
YOLOv8n	0.926	0.817	0.711	0.039
Faster R-CNN	**0.969**	**0.905**	**0.772**	0.0499
SSD300	0.948	0.754	0.623	**0.0277**

Table 1: Performance comparison of different detection models



(a) time vs mAP@0.5 (b) time vs mAP@0.5:0.95

Figure 1: Detection performance of different models

6 Code Workflow

The main workflow is summarized as follows:

1. **Library Imports and Setup**
   Essential libraries including PyTorch, Torchvision, Pandas, PIL, and COCO evaluation tools are imported. Custom visualization and conversion functions are defined early to support later stages.

2. **Data Loading and Exploration**
   The dataset is loaded from a CSV file containing bounding box annotations. Random samples are visualized using matplotlib to confirm label correctness and data quality.

```
df = pd.read_csv("annotations.csv")
sample_images = random.sample(df['image'].unique(), 5)
for img in sample_images:
    show_image_with_boxes(img, df)
```

Listing 1: Load annotations and show samples

3. **Data Splitting and Label Conversion**
   Images are split into training and validation sets using an 80/20 split. Labels are converted into two formats to support multiple models:

```
for row in df.itertuples():
    convert_to_yolo(row, save_dir="labels/train/")
coco_dict = build_coco_dict(df_train)
json.dump(coco_dict, open("coco_train.json", "w"))
```

Listing 2: Convert to YOLO and COCO format

4. **Model Training**
   Three detection models are trained separately:

- **YOLOv8**: Trained using the Ultralytics with a custom data.yaml and image folder structure.
- **Faster R-CNN**: Implemented via Torchvision with a modified classifier head to accommodate three classes.
- **SSD300**: Built with a VGG16 backbone, trained using custom data loaders.

```
model = YOLO("yolov8n.pt")
model.train(data="data.yaml", epochs=50, imgsz=256)
```

Listing 3: YOLOv8 training example

```
model = fasterrcnn_resnet50_fpn(pretrained=True)
model.roi_heads.box_predictor = FastRCNNPredictor(
    in_features, num_classes=3)
```

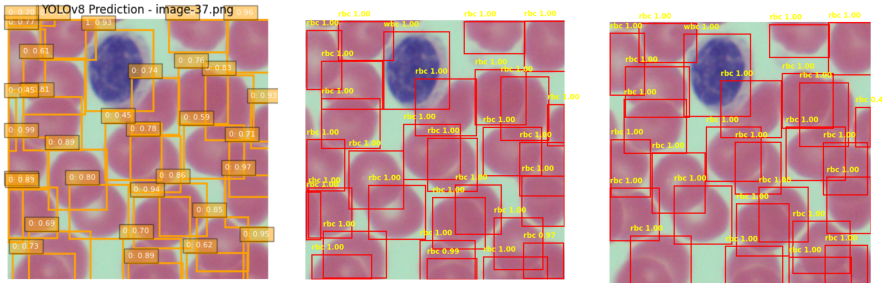
Listing 4: Faster R-CNN setup

5. **Model Evaluation**
   After training, each model generates predictions on the validation set. The results are converted to COCO format and evaluated using `pycocotools`, computing mAP metrics and average inference times.

```
eval = COCOeval(coco_gt, coco_dt, iouType='bbox')
eval.evaluate(); eval.accumulate(); eval.summarize()
```

Listing 5: Evaluate mAP using COCO tools

6. **Visualization**
   Sample predictions from each model are visualized with overlaid bounding boxes and labels. These visual results help illustrate detection quality and support the quantitative analysis.



(a) yolo v8 (b) faster r-cnn (c) ssd

Figure 2: Detection result of different models