Lab 6: Object Detection

YOLO, FCOS, and COCO Dataset Implementation

Artificial Vision (VIAR25/26)

UVigo

October 8, 2025

Section 1

Lab Overview and Objectives

Lab 6: What You'll Implement

Core Goal

Build a complete object detection system from scratch using PyTorch, implementing both YOLO-style and FCOS-style detectors, trained and evaluated on the COCO dataset.

Three Main Components

1. YOLO-Style Detector

- Grid-based detection with cell responsibility
- Multi-component loss function (localization + confidence + classification)
- Bounding box regression and class prediction

2. FCOS-Style Anchor-Free Detector

- Per-pixel prediction of (l, t, r, b) distances
- Center-ness score for quality estimation
- Feature Pyramid Network for multi-scale detection

3. Complete Training Pipeline

- COCO dataset loader with augmentation
- Focal loss for class imbalance
- Evaluation metrics (mAP, AP@IoU thresholds)
- TensorBoard logging and visualization

Learning Objectives

By the End of This Lab, You Will:

Understand:

- Difference between anchor-based (YOLO) and anchor-free (FCOS) detection
- How loss functions balance localization, confidence, and classification
- Impact of class imbalance and why focal loss helps
- Multi-scale detection via Feature Pyramid Networks

Implement:

- Complete detection pipeline from data loading to inference
- Non-maximum suppression for duplicate removal
- COCO evaluation metrics (AP, AP50, AP75, APS, APM, APL)
- Training loop with proper batch handling and gradient updates

Analyze:

- Performance trade-offs between architectures
- Impact of hyperparameters (grid size, focal loss γ, NMS threshold)
- Model behavior on different object scales

Section 2

Setup and Dataset Preparation

Environment Setup

Required Packages

```
# Create virtual environment
python -m venv venv_lab6
source venv_lab6/bin/activate  # Linux/Mac
# venv_lab6\Scripts\activate  # Windows

# Install dependencies
pip install torch torchvision
pip install pycocotools
pip install tensorboard
pip install matplotlib opencv-python
pip install tqdm numpy scipy
```

Download COCO Dataset

```
# Create data directory
mkdir -p data/coco

# Download 2017 Train/Val images (18GB train, 1GB val)
wget http://images.cocodataset.org/zips/train2017.zip
wget http://images.cocodataset.org/zips/val2017.zip

# Download annotations (241MB)
wget http://images.cocodataset.org/annotations/annotations_trainval2017.zip

# Extract
unzip train2017.zip -d data/coco/
unzip val2017.zip -d data/coco/
unzip annotations_trainval2017.zip -d data/coco/
```

Section 3

Part 1: COCO Dataset Implementation

Understanding COCO Annotations

COCO JSON Structure

```
{
  "images": [
    {
      "id": 139,
      "file_name": "000000000139.jpg",
      "height": 426,
      "width": 640
    }
  ],
  "annotations": [
    {
      "id": 1,
      "image_id": 139,
      "category_id": 1,  # 1-indexed
      "bbox": [x, y, width, height],  # XYWH format
      "area": 1234.5,
      "iscrowd": 0
    }
  ],
  "categories": [
    {
      "id": 1,
      "name": "person",
      "supercategory": "person"
    }
  ]
}
```

Key Points

- 80 object categories (IDs 1–90 with gaps)
- Boxes are **XYWH**; convert to **XYXY** for IoU
- Multiple objects per image; some images have none
- `iscrowd=1` marks crowd regions (special handling)

Dataset Loader Implementation Tasks

COCODataset Class Requirements

Initialization:

- Load JSON annotations using `pycocotools`
- Create mapping from COCO category IDs (1-90) to contiguous IDs (0-79)
- Filter images without annotations (optional)

__getitem__ method:

- Load image by ID
- Retrieve all annotations for that image
- Convert bounding boxes from XYWH to XYXY format
- Apply data augmentation (resize, flip, color jitter)
- Return: image tensor, boxes tensor, labels tensor

Collate function:

- Handle variable number of objects per image
- Stack images into batch
- Keep boxes and labels as lists (different lengths)

Expected Output Format

`image`: [3, H, W] tensor
`boxes`: [N, 4] tensor in XYXY format, normalized to [0, 1]
`labels`: [N] tensor with class IDs in range [0, 79]

Section 4

Part 2: YOLO Detector Implementation

YOLO Architecture Overview

Network Structure

Backbone: ResNet18/34/50

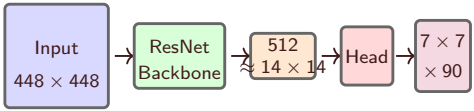
- Extract features: $\mathbb{R}^{3\times H\times W} \rightarrow \mathbb{R}^{512\times H/32\times W/32}$

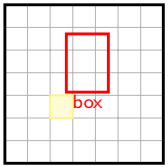
Detection Head:

- Conv layers to output grid: $S \times S \times (B \times 5 + C)$
- $S = 7$ (grid size), $B = 2$ (boxes per cell), $C = 80$ (classes)
- Output shape: $[7, 7, 90]$ for COCO

Output Interpretation:

- Each cell predicts B boxes
- Box: $(x, y, w, h, \text{conf})$ + class probs
- x, y: offsets relative to cell (0-1)
- w, h: relative to image (0-1)





$S \times S$ Grid (S=7)

YOLO Loss Function Components

Multi-Part Loss: $\mathcal{L} = \lambda_{coord}\mathcal{L}_{coord} + \mathcal{L}_{conf}^{obj} + \lambda_{noobj}\mathcal{L}_{conf}^{noobj} + \mathcal{L}_{class}$

1. Localization Loss ($\mathcal{L}_{coord}$):

$$\mathcal{L}_{coord} = \sum_{i=0}^{S^2}\sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 + (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2\right]$$

2. Confidence Loss (Object):

$$\mathcal{L}_{conf}^{obj} = \sum_{i=0}^{S^2}\sum_{j=0}^{B} \mathbb{1}_{ij}^{obj}(C_i - \hat{C}_i)^2$$

3. Confidence Loss (No Object):

$$\mathcal{L}_{conf}^{noobj} = \sum_{i=0}^{S^2}\sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj}(C_i - \hat{C}_i)^2$$

4. Classification Loss:

$$\mathcal{L}_{class} = \sum_{i=0}^{S^2} \mathbb{1}_{i}^{obj} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2$$

Typical Hyperparameters

$\lambda_{coord} = 5.0$, $\lambda_{noobj} = 0.5$, $S = 7$, $B = 2$

YOLO Implementation Tasks

Step 1: Encode Ground Truth to Grid

For each bounding box:

- Find which grid cell contains the center
- Compute offsets (x, y) relative to cell (0-1)
- Set target confidence $= 1$ for responsible box
- Set class probabilities for that cell

Step 2: Forward Pass

```
# Input: [B, 3, 448, 448]
# Backbone output: [B, 512, 14, 14]
# Detection head output: [B, S, S, B*5+C]
# Reshape to: [B, S, S, B, 5+C]
# where last dim = [x, y, w, h, conf, class_probs...]
```

Step 3: Compute Loss

- Determine responsible boxes (highest IoU with GT)
- Apply $\mathbb{1}_{ij}^{obj}$ and $\mathbb{1}_{ij}^{noobj}$ masks
- Calculate each loss component
- Weight and sum: $\mathcal{L}_{total}$

Section 5

Part 3: FCOS Anchor-Free Detector

FCOS Architecture and Predictions

Per-Pixel Predictions

At each feature map location (i, j):

Classification: C scores

$$\mathbf{p}_{ij} \in \mathbb{R}^C$$

Regression: (l, t, r, b) distances

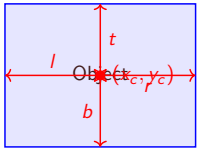
$$\mathbf{r}_{ij} = (l_{ij}, t_{ij}, r_{ij}, b_{ij}) \in \mathbb{R}^4$$

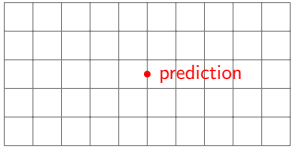
Center-ness: Quality score

$$s_{ij} = \sqrt{\frac{\min(l, r)}{\max(l, r)} \times \frac{\min(t, b)}{\max(t, b)}}$$

Key Difference from YOLO

- No grid-based assignment
- Direct per-pixel prediction
- Multi-scale via FPN levels





Feature Map

Center-ness Intuition:

- High near object center
- Low at boundaries
- Suppresses low-quality boxes

Feature Pyramid Network (FPN) for Multi-Scale

FPN Construction

Bottom-up pathway: ResNet stages

$$C_2, C_3, C_4, C_5 \text{ with strides } 4, 8, 16, 32$$

Top-down pathway: Lateral connections

$$P_5 = \text{Conv}_{1\times1}(C_5)$$
$$P_4 = \text{Conv}_{3\times3}(\text{Upsample}(P_5) + \text{Conv}_{1\times1}(C_4))$$
$$P_3 = \text{Conv}_{3\times3}(\text{Upsample}(P_4) + \text{Conv}_{1\times1}(C_3))$$

Scale assignment:

- P_3: objects with $\max(l, t, r, b) \in [0, 64]$ pixels
- P_4: objects with $\max(l, t, r, b) \in [64, 128]$ pixels
- P_5: objects with $\max(l, t, r, b) > 128$ pixels

Implementation Note

Each FPN level has independent classification and regression heads (shared weights across spatial locations).

FCOS Loss Function

Three Components

1. Classification Loss (Focal Loss):

$$\mathcal{L}_{cls} = -\frac{1}{N_{pos}} \sum_{x,y} \alpha_t (1 - p_t)^{\gamma} \log(p_t)$$

where $\gamma = 2.0$, $\alpha = 0.25$

2. Regression Loss (GIoU):

$$\mathcal{L}_{reg} = \frac{1}{N_{pos}} \sum_{x,y} \mathbb{1}_{\{c^*_{x,y} > 0\}} \mathcal{L}_{GIoU}(b_{x,y}, b^*_{x,y})$$

3. Center-ness Loss (BCE):

$$\mathcal{L}_{centerness} = \frac{1}{N_{pos}} \sum_{x,y} \mathbb{1}_{\{c^*_{x,y} > 0\}} \text{BCE}(s_{x,y}, s^*_{x,y})$$

Total Loss:

$$\mathcal{L} = \mathcal{L}_{cls} + \lambda_1 \mathcal{L}_{reg} + \lambda_2 \mathcal{L}_{centerness}$$

Typical: $\lambda_1 = 1.0$, $\lambda_2 = 1.0$

FCOS Implementation Tasks

Step 1: Assign Ground Truth to Locations

```
for each ground truth box:
    for each FPN level P_k:
        for each location (x, y) in P_k:
            # Compute (l, t, r, b) from location to box
            l = x - bbox.x1
            t = y - bbox.y1
            r = bbox.x2 - x
            b = bbox.y2 - y

            # Check if inside box and within scale range
            if l > 0 and t > 0 and r > 0 and b > 0:
                if min_scale < max(l,t,r,b) < max_scale:
                    # Assign as positive sample
                    targets[k][y, x] = (l, t, r, b, class)
```

Step 2: Compute Center-ness Targets

```
centerness = sqrt(
    (min(l, r) / max(l, r)) *
    (min(t, b) / max(t, b))
)
```

Section 6

Part 4: Loss Functions and Training

Focal Loss Implementation

Purpose: Address Class Imbalance

In object detection, most locations are background (negative samples). Focal loss down-weights easy examples and focuses on hard ones.

Mathematical Formulation

Standard Cross-Entropy:

$$CE(p_t) = -\log(p_t)$$

Focal Loss:

$$FL(p_t) = -\alpha_t(1 - p_t)^\gamma \log(p_t)$$

where:

- $p_t = p$ if $y = 1$ (positive), else $p_t = 1 - p$
- $\gamma \geq 0$ is focusing parameter (typical: 2)
- α_t is class weighting (typical: 0.25 for positive)

Behavior

- When $p_t \to 1$ (easy example): $(1 - p_t)^\gamma \to 0$, loss $\to 0$
- When $p_t \to 0$ (hard example): $(1 - p_t)^\gamma \to 1$, loss remains high
- γ controls the rate of down-weighting

Non-Maximum Suppression (NMS)

Purpose

Remove duplicate detections for the same object by keeping only the highest confidence box among overlapping predictions.

Algorithm

```python
def nms(boxes, scores, iou_threshold=0.5):
    # Sort by confidence descending
    indices = scores.argsort(descending=True)
    keep = []

    while len(indices) > 0:
        # Keep highest confidence box
        current = indices[0]
        keep.append(current)

        # Compute IoU with remaining boxes
        ious = compute_iou(boxes[current], boxes[indices[1:]])

        # Remove overlapping boxes
        mask = ious <= iou_threshold
        indices = indices[1:][mask]

    return keep
```

Key Parameters

`iou_threshold`: Higher $\Rightarrow$ more boxes kept (default: 0.5)

Training Pipeline Structure

Main Training Loop

```python
for epoch in range(num_epochs):
    model.train()

    for images, boxes, labels in train_loader:
        # Forward pass
        predictions = model(images)

        # Compute loss
        loss = compute_loss(predictions, boxes, labels)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Logging
        if step % log_interval == 0:
            writer.add_scalar('train/loss', loss.item(), step)

    # Validation
    if epoch % val_interval == 0:
        metrics = evaluate(model, val_loader)
        writer.add_scalar('val/mAP', metrics['mAP'], epoch)

    # Save checkpoint
    save_checkpoint(model, optimizer, epoch)
```

Section 7

Part 5: Evaluation Metrics

COCO Evaluation Metrics

Primary Metrics

AP (Average Precision): Mean AP over IoU thresholds [0.5:0.05:0.95]

$$\text{mAP} = \frac{1}{10} \sum_{t=0.5}^{0.95} \text{AP}_t$$

AP50: AP at IoU threshold 0.5 (PASCAL VOC metric)
AP75: AP at IoU threshold 0.75 (stricter localization)

Scale-Specific Metrics

- **APS:** Small objects (area $< 32^2$ pixels)
- **APM:** Medium objects ($32^2 \leq$ area $< 96^2$)
- **APL:** Large objects (area $\geq 96^2$)

Implementation

Use `pycocotools.cocoeval.COCOeval` for official evaluation:

- Matches predictions to ground truth via IoU
- Computes precision-recall curves
- Aggregates across categories and IoU thresholds

Evaluation Code Structure

Evaluation Function

```python
def evaluate(model, data_loader, device):
    model.eval()
    coco_results = []

    with torch.no_grad():
        for images, image_ids in data_loader:
            # Forward pass
            predictions = model(images.to(device))

            # Post-process: NMS
            detections = post_process(predictions)

            # Convert to COCO format
            for det, img_id in zip(detections, image_ids):
                for box, score, label in det:
                    coco_results.append({
                        'image_id': img_id,
                        'category_id': label,
                        'bbox': [x, y, w, h],  # XYWH
                        'score': score
                    })

    # Evaluate using COCO API
    coco_eval = COCOeval(coco_gt, coco_dt, 'bbox')
    coco_eval.evaluate()
    coco_eval.accumulate()
    coco_eval.summarize()

    return {
        'mAP': coco_eval.stats[0],
        'AP50': coco_eval.stats[1],
        'AP75': coco_eval.stats[2]
    }
```

Section 8

Implementation Roadmap

Step-by-Step Implementation Guide

Phase 1: Data Pipeline (2-3 hours)

1. Implement `COCODataset` class with annotation parsing
2. Add data augmentation (resize, flip, normalize)
3. Create custom collate function for variable-length boxes
4. Test: Visualize loaded images with ground truth boxes

Phase 2: YOLO Detector (3-4 hours)

1. Build YOLO model with ResNet backbone
2. Implement grid-based target encoding
3. Write YOLO loss function with all components
4. Test: Forward pass and loss computation

Phase 3: FCOS Detector (3-4 hours)

1. Implement FPN for multi-scale features
2. Build FCOS heads (classification, regression, centerness)
3. Write focal loss and GIoU loss
4. Implement target assignment logic

Step-by-Step Implementation Guide (cont.)

Phase 4: Training Pipeline (2-3 hours)

1. Set up training loop with optimizer and scheduler
2. Add TensorBoard logging for losses and metrics
3. Implement checkpoint saving and loading
4. Start training and monitor convergence

Phase 5: Evaluation and Post-Processing (2-3 hours)

1. Implement NMS for inference
2. Write evaluation function using COCO API
3. Compute mAP, AP50, AP75, and scale-specific metrics
4. Visualize predictions on validation set

Phase 6: Analysis and Report (2 hours)

1. Compare YOLO vs FCOS performance
2. Analyze scale-specific performance (APS, APM, APL)
3. Experiment with hyperparameters (learning rate, loss weights)
4. Write report with findings and visualizations

Section 9

Tips and Common Pitfalls

Implementation Tips

Data Loading

- **Memory:** Use `num_workers > 0` in DataLoader for parallel loading
- **Normalization:** Apply ImageNet mean/std for pretrained backbones
- **Augmentation:** Strong augmentation helps (flip, color jitter, crop)

Training Stability

- **Gradient clipping:** Use `torch.nn.utils.clip_grad_norm_()` (max norm $= 10$)
- **Warmup:** Start with low learning rate for first few epochs
- **Loss weighting:** Balance localization vs classification losses
- **Batch size:** Larger is better (8-16 on GPUs), use gradient accumulation if needed

Debugging

- **Overfitting check:** Train on small subset (10 images) first
- **Visualization:** Plot predictions vs ground truth frequently
- **NaN losses:** Check for division by zero, use `torch.clamp()`
- **IoU computation:** Ensure boxes are in correct format (XYXY vs XYWH)

Common Pitfalls and Solutions

Problem: Low mAP (¡10%)

Possible Causes:

- Incorrect target encoding (grid assignment, box format)
- Wrong loss weighting (confidence loss too dominant)
- NMS threshold too aggressive

Solutions: Visualize targets vs predictions, check loss components individually

Problem: Training Doesn't Converge

Possible Causes:

- Learning rate too high (try 1e-4 instead of 1e-3)
- Missing gradient clipping
- Imbalanced loss components

Solutions: Reduce LR, add warmup, check loss curves in TensorBoard

Problem: Good Training Loss, Poor Validation

Cause: Overfitting or domain shift
Solutions: More augmentation, dropout, smaller model, regularization

Expected Results and Benchmarks

Target Performance (after 20 epochs)

YOLO (Grid 7×7, ResNet18 backbone):

- mAP: 15-20% (lightweight, fast)
- AP50: 30-35%
- Inference: 30-40 FPS on GPU

FCOS (FPN, ResNet50 backbone):

- mAP: 25-30% (better than YOLO)
- AP50: 42-48%
- APS: 10-15% (improved small object detection)
- Inference: 15-20 FPS on GPU

Note

These are expected results with limited training. State-of-the-art detectors achieve 40-50+ mAP with longer training, larger backbones, and advanced techniques.

GPU Requirements

- Minimum: 6GB VRAM (batch size 4-8)
- Recommended: 11GB+ VRAM (batch size 16)
- CPU training possible but 10-20× slower

Section 10

Deliverables and Grading

Lab Deliverables

Required Submissions

1. **Code (60%)**
   - Complete implementation of YOLO and FCOS detectors
   - COCO dataset loader with augmentation
   - Training and evaluation scripts
   - All code must run without errors
2. **Trained Models (20%)**
   - Checkpoint files for both YOLO and FCOS
   - Training logs (TensorBoard events)
   - Minimum 15 epochs of training
3. **Report (20%)**
   - Architecture descriptions with diagrams
   - Training curves (loss, mAP over epochs)
   - Evaluation results table (mAP, AP50, AP75, APS, APM, APL)
   - Comparison analysis: YOLO vs FCOS
   - Example predictions (5-10 images with visualizations)
   - Discussion of failures and potential improvements

Grading Rubric

Detailed Breakdown

Code Quality (60 points):

- Dataset implementation: 10 pts
- YOLO model and loss: 15 pts
- FCOS model and loss: 15 pts
- Training pipeline: 10 pts
- Evaluation and NMS: 10 pts

Model Performance (20 points):

- YOLO mAP $> 15\%$: 8 pts (partial credit for 10-15%)
- FCOS mAP $> 25\%$: 12 pts (partial credit for 20-25%)

Report Quality (20 points):

- Completeness and clarity: 8 pts
- Results presentation: 6 pts
- Analysis depth: 6 pts

Bonus Points (up to 10%)

Additional features: Soft-NMS, DIoU loss, data augmentation experiments, etc.

Section 11

Resources and Support

Helpful Resources

Documentation

- **PyTorch**: `https://pytorch.org/docs/`
- **COCO API**: `https://github.com/cocodataset/cocoapi`
- **Torchvision**: `https://pytorch.org/vision/stable/index.html`

Papers

- **YOLO**: Redmon et al., "You Only Look Once" (CVPR 2016)
- **FCOS**: Tian et al., "FCOS: Fully Convolutional One-Stage" (ICCV 2019)
- **FPN**: Lin et al., "Feature Pyramid Networks" (CVPR 2017)
- **Focal Loss**: Lin et al., "Focal Loss for Dense Object Detection" (ICCV 2017)

Reference Implementations

- Ultralytics YOLO: `https://github.com/ultralytics/yolov5`
- MMDetection: `https://github.com/open-mmlab/mmdetection`
- Detectron2: `https://github.com/facebookresearch/detectron2`