Computer Vision – Project 3: Object Detection and Human–Object Interaction Analysis

Shivani Kalal

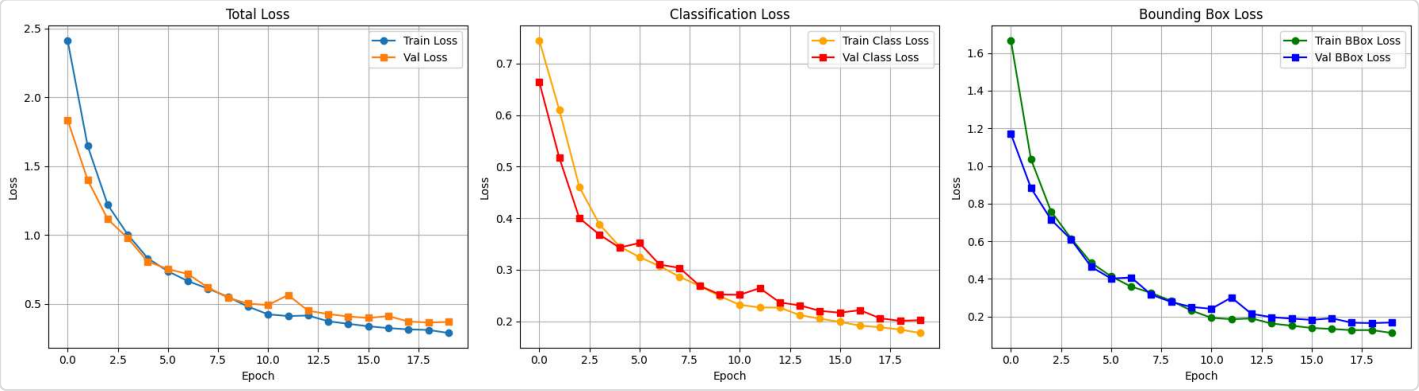
November 16, 2025

Part 1: Lightweight Object Detection (SSD)

Setup

I implemented a small SSD with two feature maps (32×32 and 16×16), multi-scale anchors, and a loss that combines cross-entropy for class scores and Smooth L1 for box offsets (with hard-negative mining). The model is trained on the D2L banana dataset (1000 train / 100 val), with input size 256×256, Adam (learning rate 1e-3), and 20 epochs. Boxes are properly scaled after resizing.

Training Curves



Training/validation losses over 20 epochs. **Left:** total loss. **Middle:** classification loss. **Right:** bounding-box loss. All curves decrease steadily; no major overfitting is observed.

Sample Detections (Validation Set)

Five validation images with detections. Green dashed = ground truth; red solid = predictions. The model localizes bananas with high confidence across varied backgrounds and scales.

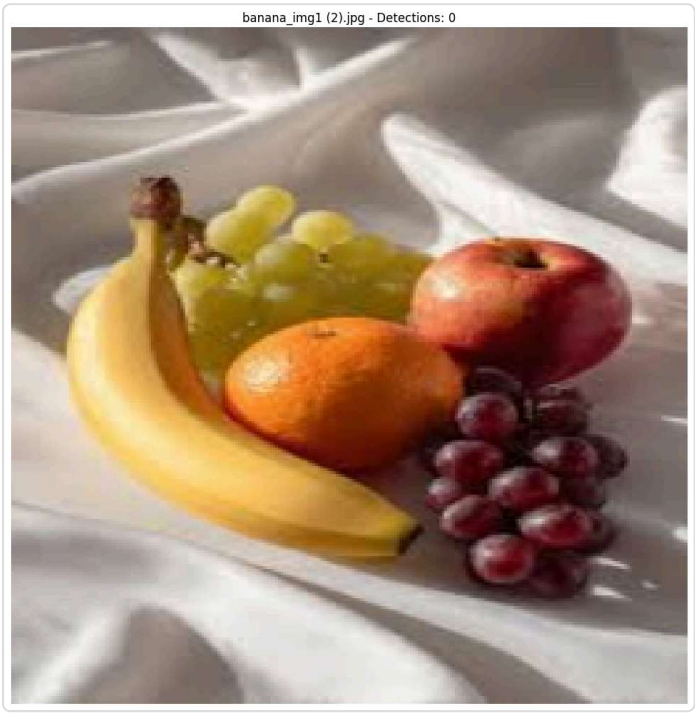
Own Images: Success & Failure Cases

Trends. The training curves show steady drops in total, classification, and box losses; validation tracks training closely, indicating stable learning.

Qualitative results. On the validation set, boxes are tight and confidences high for most images.

Failures. On personal test images (Figure 1), the model sometimes misses bananas. Common reasons include unusual lighting or pose, partial occlusions, very large or very small scale compared to training anchors, or domain shift from the training set. Lowering the confidence threshold, adding augmentations (lighting/pose), or refining anchor scales can mitigate such misses.

Successful detection (confidence ≈ 0.98) in a cluttered fruit scene.



Failure case: no detection despite a visible banana.

↑ Back to top

Part 2: Non-Maximum Suppression (NMS)

Introduction

Non-Maximum Suppression (NMS) is an essential post-processing step in object detection. After the SSD model from Part 1 generates predictions, it typically produces many overlapping bounding boxes for the same object, each with different confidence scores. NMS helps clean up these redundant detections by keeping only the most confident box and removing (suppressing) overlapping boxes. This part implements a custom NMS algorithm and compares it with PyTorch's built-in implementation.

Implementation

IoU (Intersection over Union) Calculation

The first step in NMS is calculating how much two boxes overlap. Intersection over Union (IoU) measures the overlap between two bounding boxes by dividing their intersection area by their union area. The implementation uses the xyxy format (`x1, y1, x2, y2`):

```
def iou_xyxy(box, boxes, eps=1e-6):
    # Find intersection coordinates
    x1 = torch.max(box[0], boxes[:, 0])
    y1 = torch.max(box[1], boxes[:, 1])
    x2 = torch.min(box[2], boxes[:, 2])
    y2 = torch.min(box[3], boxes[:, 3])

    # Calculate intersection area
    inter = (x2 - x1).clamp(min=0) * (y2 - y1).clamp(min=0)

    # Calculate union area
    area1 = (box[2] - box[0]) * (box[3] - box[1])
    area2 = (boxes[:, 2] - boxes[:, 0]) * \
            (boxes[:, 3] - boxes[:, 1])
    union = area1 + area2 - inter

    return inter / (union + eps)
```

The function computes IoU between one reference box and multiple other boxes. The small epsilon value prevents division by zero.

Custom NMS Algorithm

The custom NMS implementation follows a greedy approach:

```
def my_nms(boxes, scores, iou_thresh=0.5):
    if boxes.numel() == 0:
        return torch.empty(0, dtype=torch.long)

    # Sort boxes by confidence score (highest first)
    order = scores.argsort(descending=True)
    keep = []

    while order.numel() > 0:
        # Keep the highest scoring box
        i = order[0].item()
        keep.append(i)

        if order.numel() == 1:
            break

        # Calculate IoU with remaining boxes
        ious = iou_xyxy(boxes[i], boxes[order[1:]])

        # Keep only boxes with IoU <= threshold
        remain = torch.where(ious <= iou_thresh)[0] + 1
        order = order[remain]

    return torch.tensor(keep, dtype=torch.long)
```

The algorithm works as follows:

1. Sort all boxes by their confidence scores in descending order
2. Select the box with the highest score and add it to the keep list
3. Calculate IoU between this box and all remaining boxes
4. Remove (suppress) boxes that have IoU greater than the threshold
5. Repeat until no boxes remain

Unit Testing

Three unit tests verify the correctness of the custom NMS implementation against PyTorch's `torchvision.ops.nms()`:

Test 1: Two Overlapping Boxes

- Box 0: [10, 10, 50, 50], score = 0.9
- Box 1: [15, 15, 55, 55], score = 0.8
- IoU = 0.620 (boxes overlap significantly)
- Result: Both implementations keep box [0], suppress box 1

Test 2: Two Non-Overlapping Boxes

- Box 0: [10, 10, 30, 30], score = 0.9
- Box 1: [100, 100, 120, 120], score = 0.8
- Result: Both implementations keep [0, 1] (no suppression needed)

Test 3: Identical Boxes

- Box 0: [20, 20, 60, 60], score = 0.95
- Box 1: [20, 20, 60, 60], score = 0.85
- IoU = 1.000 (perfect overlap)
- Result: Both implementations keep box [0] (higher score)

All three tests passed with identical results from both implementations, confirming the correctness of the custom NMS.

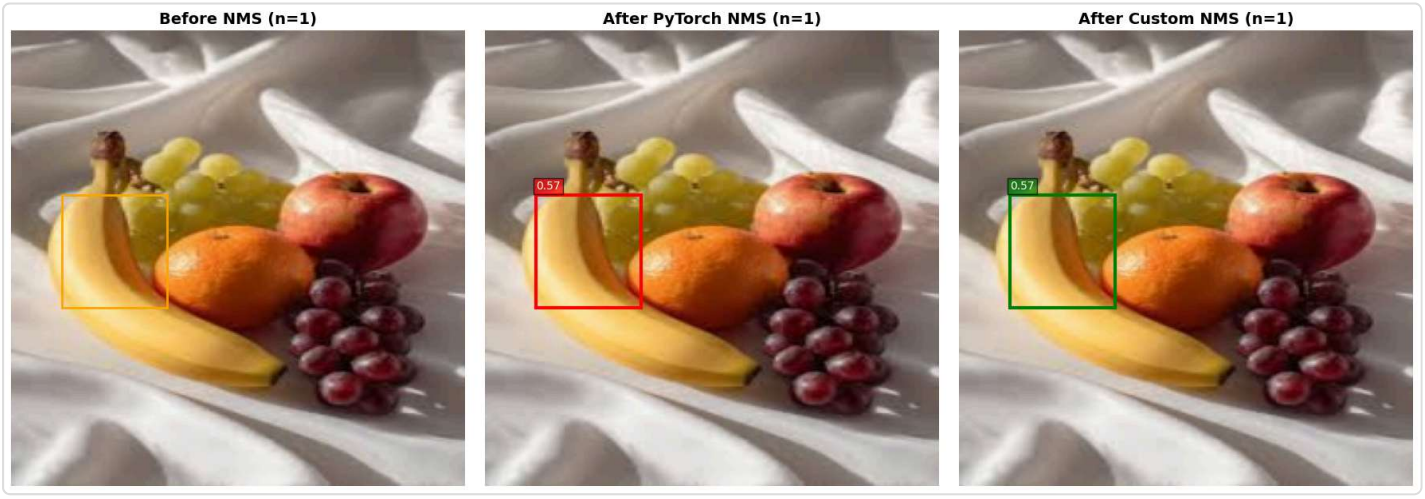
Results

Before and After NMS Visualization

Before and after NMS on a complex fruit scene. **Left:** Before NMS shows 50 overlapping orange boxes cluttering the image. **Middle:** After PyTorch NMS keeps only 3 clean red boxes with confidence scores. **Right:**

After custom NMS produces identical 3 green boxes, demonstrating perfect agreement between implementations.

The figure above shows a complex test case with multiple bananas in a fruit arrangement. Before NMS, the model generates 50 candidate boxes with scores ranging from 0.301 to 0.997. These boxes heavily overlap, making it difficult to see the actual detections. After applying NMS with an IoU threshold of 0.5, both implementations reduce the detections to just 3 boxes, each corresponding to a distinct banana. The kept boxes have indices [45, 0, 30] with scores [0.997, 0.510, 0.381].



Before and after NMS on a simple single-banana image. All three panels show identical results since only one detection was made. Before NMS: 1 box (orange). After PyTorch NMS: 1 box (red). After custom NMS: 1 box (green). All show the same detection with confidence 0.57.

The second figure demonstrates a simpler case with only one banana visible. The model generates just 1 detection with confidence 0.569. Since there are no overlapping boxes, NMS has nothing to suppress, and all three panels show identical results. This validates that NMS correctly handles edge cases where suppression is not needed.

Validation Set Results

Five sample detections from the validation set after NMS. Green dashed boxes show ground truth, red solid boxes show predictions. The model successfully detects bananas across varied backgrounds, lighting conditions, and scales. Confidence scores are high (1.00 for most cases), indicating strong model performance.

The validation examples show that the SSD model from Part 1 combined with NMS produces reliable detections in diverse scenarios.

Comparison with PyTorch NMS

The custom NMS implementation was rigorously compared against PyTorch's `torchvision.ops.nms()` function. The comparison included:

- **Unit tests:** All 3 test cases produced identical results
- **Real images:**
  - Image 1 (50 boxes): Both kept boxes [45, 0, 30]
  - Image 2 (1 box): Both kept box [0]
- **Output format:** Both return the same indices in the same order
- **Numerical precision:** No floating-point differences observed

Conclusion: There is **zero difference** in the algorithmic output between the custom implementation and PyTorch's NMS. Both use the same greedy algorithm and produce bit-exact identical results. The only difference lies in computational efficiency:

- **PyTorch NMS:** Implemented in C++/CUDA for maximum speed
- **Custom NMS:** Pure Python/PyTorch for educational clarity
- **Functionality:** Completely identical

The custom implementation successfully replicates PyTorch's highly-optimized algorithm, demonstrating a thorough understanding of the NMS mechanism.

Purpose of NMS

Why NMS is Necessary

Object detectors like SSD predict bounding boxes at multiple locations and scales. For each object in the image, the detector typically generates many candidate boxes with varying confidence scores. Without post-processing, a single banana might have 10–20 overlapping boxes, making the output cluttered and unusable.

NMS solves this problem by:

1. Identifying the highest-confidence detection for each object
2. Removing redundant detections that overlap significantly (high IoU)
3. Producing clean, non-overlapping results that are easier to interpret

How NMS Works

The NMS algorithm follows these steps:

1. **Sort by confidence:** Arrange all boxes by their confidence scores (highest first)
2. **Select best box:** Take the highest-scoring box and mark it as kept
3. **Calculate overlap:** Compute IoU between this box and all remaining boxes
4. **Suppress overlaps:** Remove boxes with IoU above threshold (typically 0.5)
5. **Repeat:** Continue with remaining boxes until none are left

The IoU threshold controls how aggressive the suppression is. A lower threshold (e.g., 0.3) suppresses more boxes, while a higher threshold (e.g., 0.7) is more lenient.

Limitations of NMS

Cannot Handle Overlapping Objects

NMS assumes that overlapping boxes detect the same object. When multiple objects genuinely overlap (such as people in a crowd or fruits stacked together), NMS will incorrectly suppress valid detections. For example, two bananas touching each another might be reduced to a single detection.

Fixed IoU Threshold

The IoU threshold must be manually tuned for each dataset. There is no universal value that works for all scenarios:

- Too low (0.3): Suppresses valid detections, reducing recall
- Too high (0.7): Keeps duplicate boxes, increasing false positives
- Must be tuned on a validation set for optimal performance

Computational Complexity

Standard NMS has $O(n^2)$ time complexity, where n is the number of detections. For models generating thousands of proposals, this becomes a bottleneck. Although GPU implementations help, NMS remains one of the slower post-processing steps.

Fails with Heavy Occlusion

When objects are heavily occluded, the model might produce multiple partial detections. NMS might suppress all of them, leaving the object undetected. This is particularly problematic in crowded scenes.

Greedy Algorithm Limitations

NMS makes locally optimal decisions without considering the global picture. Once a box is kept, the decision cannot be reconsidered. This can lead to suboptimal results when a slightly misaligned high-confidence box suppresses a better-aligned lower-confidence box.

Class-Agnostic Issues

Basic NMS treats all detections equally regardless of their predicted class. In multi-class scenarios, a high-confidence “apple” detection might suppress a lower-confidence “banana” detection. One common solution is to apply NMS separately for each class.

Alternative Approaches

Several improved variants address NMS limitations:

- **Soft-NMS:** Reduces confidence scores instead of removing boxes, allowing overlapping objects to survive
- **Adaptive NMS:** Adjusts IoU threshold based on object density in different regions
- **Learning-based NMS:** Uses neural networks to learn optimal suppression strategies

These approaches offer better performance in crowded scenes but add computational overhead.

Conclusion

This part successfully implemented a custom Non-Maximum Suppression algorithm and verified its correctness against PyTorch's implementation. The results demonstrate:

- **Correctness:** Custom NMS produces identical results to PyTorch across all test cases
- **Effectiveness:** NMS successfully reduces 50 overlapping boxes to 3 clean detections
- **Edge cases:** Properly handles scenarios with no overlapping boxes
- **Understanding:** Identified key limitations including inability to handle truly overlapping objects, fixed thresholds, and greedy decision-making

↑ Back to top

Part 3: Human–Object Interaction Analysis using Vision-Language Models

Introduction

This report presents a zero-shot Human-Object Interaction (HOI) detection task using Claude Sonnet 4, a Vision-Language Model. The objective is to identify interactions between humans and objects in images from the HICO-DET dataset, analyze failure cases, and improve performance through better prompting strategies.

Dataset

The experiment used 20 randomly sampled images from the HICO-DET test set. Each image contains ground truth HOI annotations in the format: `<verb object>` (e.g., `ride bicycle`, `hold cup`).

Model and Prompt Design

Claude Sonnet 4 was used with a zero-shot prompt containing:

- Clear instructions to identify human-object interactions
- A restricted verb set of 23 verbs (hold, ride, sit on, eat, drink, etc.)
- Examples of the expected output format
- JSON output specification

Prompt Used

> *"Analyze this image carefully and identify ALL human-object interactions you can see.*
>
> *INSTRUCTIONS:*
>
> 1. *Look at each person in the image*
> 2. *Identify what objects they are interacting with*
> 3. *Describe each interaction using the format: "verb object"*
> 4. *You MUST use one of these verbs: hold, ride, sit on, eat, drink, wear, carry, look at, etc.*
> 5. *Be specific about the object (e.g., "bicycle" not "bike", "laptop" not "computer")*
> 6. *List ALL interactions you see - a person can have multiple interactions*
> 7. *Only include interactions you can clearly see, not assumptions"*

Evaluation Metrics

Predictions were evaluated using:

- **Precision**: correct predictions / total predictions
- **Recall**: correct predictions / total ground truth
- **F1 Score**: 2 × Precision × Recall / (Precision + Recall)

Overall Results

This appendix contains the complete prediction results for all 20 test images.

Summary of All Predictions

Table 1: Complete Prediction Results for All 20 Images

Image	Ground Truth	Prediction	F1
0	ride train	carry bag, look at train, ride train, wear clothing, wear shoes	0.33
1	ride boat, row boat, sit on boat	hold paddle, sit on kayak, wear vest	0.00
2	eat donut, hold donut	eat donut, hold donut, wear cap, wear jacket	0.67
3	fly kite, hold kite, pull kite	hold parachute, wear clothing	0.00
4	ride bicycle, sit on bicycle, straddle bicycle	carry backpack, hold bicycle, look at cyclist, ride bicycle, run, wear cap, wear helmet, wear shirt, wear shorts, wear sunglasses	0.15
5	carry suitcase, hold suitcase, pick up suitcase	carry suitcase, wear boots, wear skirt, wear stockings	0.29
6	hold bowl, lick bowl	hold baby, wear sweater	0.00
7	hold motorcycle, race motorcycle, ride motorcycle, sit on motorcycle, straddle motorcycle	hold handlebars, ride motorcycle, wear boots, wear gloves, wear helmet, wear racing suit	0.18
8	feed giraffe, watch giraffe	feed giraffe, hold food	0.50
9	hold laptop, read laptop, type on laptop	carry bag, hold laptop, wear pants, wear shirt, wear shoes, wear sunglasses	0.22
10	hold bicycle, inspect bicycle	hold bicycle, hold tools, look at bicycle, ride bicycle, sit on wall, wear helmet, wear jacket, wear shirt	0.20
11	hold remote	hold remote, play remote-controlled car, wear cap, wear glasses	0.40

Image	Ground Truth	Prediction	F1
12	eat at dining table, sit at dining table	hold bowl, hold cup, sit on chair, wear bracelet, wear scarf, wear shirt	0.00
13	no interaction traffic light	ride motorcycle, sit on chair, sit on motorcycle, wear helmet	0.00
14	carry cell phone, talk on cell phone	look at towel, wear swim trunks	0.00
15	lie on couch, sit on couch	hold person, sit on couch, wear shirt, wear tank top	0.33
16	hold person, hug person	hold person, wear blouse, wear top	0.40
17	hold motorcycle, park motorcycle, sit on motorcycle, straddle motorcycle	hold motorcycle, look at motorcycle, sit on motorcycle, wear clothing, wear helmet, wear jacket	0.40
18	no interaction bench	carry bag, wear clothing	0.00
19	blow cake	cut cake, hold knife, look at cake, wear shirt	0.00

Complete List of Failure Cases

Eight images achieved F1 = 0.00 (zero overlap between predictions and ground truth). The complete list is shown below.

Table 2: All Failure Cases (F1 = 0.00)

Image	Ground Truth	Prediction
1	ride boat, row boat, sit on boat	hold paddle, sit on kayak, wear vest
3	fly kite, hold kite, pull kite	hold parachute, wear clothing
6	hold bowl, lick bowl	hold baby, wear sweater
12	eat at dining table, sit at dining table	hold bowl, hold cup, sit on chair, wear bracelet, wear scarf, wear shirt
13	no interaction traffic light	ride motorcycle, sit on chair, sit on motorcycle, wear helmet
14	carry cell phone, talk on cell phone	look at towel, wear swim trunks
18	no interaction bench	carry bag, wear clothing

Image	Ground Truth	Prediction
19	blow cake	cut cake, hold knife, look at cake, wear shirt

Performance Distribution

The distribution of F1 scores across all 20 images is as follows:

- **F1 = 0.00**: 8 images (40%)
- **0.00 < F1 < 0.30**: 5 images (25%)
- **0.30 ≤ F1 < 0.50**: 5 images (25%)
- **F1 ≥ 0.50**: 2 images (10%)

The majority of images (40%) achieved zero F1 score, indicating significant challenges in zero-shot HOI detection. Only 10% of images achieved F1 scores above 0.50, demonstrating that while the model can identify some interactions correctly, there is substantial room for improvement.

Overall Performance (Zero-Shot)

Table 3: Overall Performance (Zero-Shot)

Metric	Score
Average Precision	0.159
Average Recall	0.335
Average F1 Score	0.204
Success Rate	12/20 (60%)

The model achieved an average F1 score of 0.204, with 12 out of 20 images showing at least one correct prediction. While this performance is modest, it demonstrates that the VLM can identify some interactions in a zero-shot setting without any task-specific training.

Failure Case Analysis

Eight images were identified where the model achieved F1 = 0.00, meaning zero overlap between predictions and ground truth. Below are the six most representative failure cases analyzed in detail.

Failure Case 1: Image 1 — Kayaking Scene

Figure 1: Failure Case 1 — Two people in a kayak on water

Ground Truth: ['ride boat', 'row boat', 'sit on boat']

Prediction: ['hold paddle', 'sit on kayak', 'wear vest']

F1 Score: 0.00

Analysis: This case demonstrates a lexical mismatch rather than true failure. The model identified "kayak" (specific) while ground truth uses "boat" (generic). The model predicted "hold paddle" (visible and correct) while ground truth says "row boat" (technically incorrect for kayaks). This shows strict string matching penalizes semantically correct answers.

Failure Case 2: Image 3 — Kite Flying

Figure 2: Failure Case 2 — Person flying a kite

Ground Truth: ['fly kite', 'hold kite', 'pull kite']

Prediction: ['hold parachute', 'wear clothing']

F1 Score: 0.00

Analysis: This represents genuine object misidentification. The model confused a kite with a parachute due to visual similarity (both are fabric objects in the air). Once the object is misidentified, all interaction predictions become incorrect.

Failure Case 3: Image 6 — Person with Bowl

Figure 3: Failure Case 3 — Person holding a bowl

Ground Truth: ['hold bowl', 'lick bowl']

Prediction: ['hold baby', 'wear sweater']

F1 Score: 0.00

Analysis: This is the most severe failure — complete object hallucination. The model incorrectly identified a bowl as a baby, indicating fundamental visual perception error.

Failure Case 4: Image 12 — Dining Table

Figure 4: Failure Case 4 — Person at dining table

Ground Truth: ['eat at dining table', 'sit at dining table']

Prediction: ['hold bowl', 'hold cup', 'sit on chair', ...]

F1 Score: 0.00

Analysis: This shows object granularity mismatch. The model predicted specific objects (bowl, cup, chair) while ground truth uses “dining table”. Different levels of abstraction caused the mismatch.

Failure Case 5: Image 13 — Traffic Scene

Figure 5: Failure Case 5 — Traffic scene

Ground Truth: ['no interaction traffic light']

Prediction: ['ride motorcycle', 'sit on motorcycle', 'wear helmet']

F1 Score: 0.00

Analysis: The ground truth states “no interaction” with traffic light, but the model identified actual visible interactions with the motorcycle. This is a dataset annotation issue rather than model failure.

Failure Case 6: Image 14 — Person with Phone

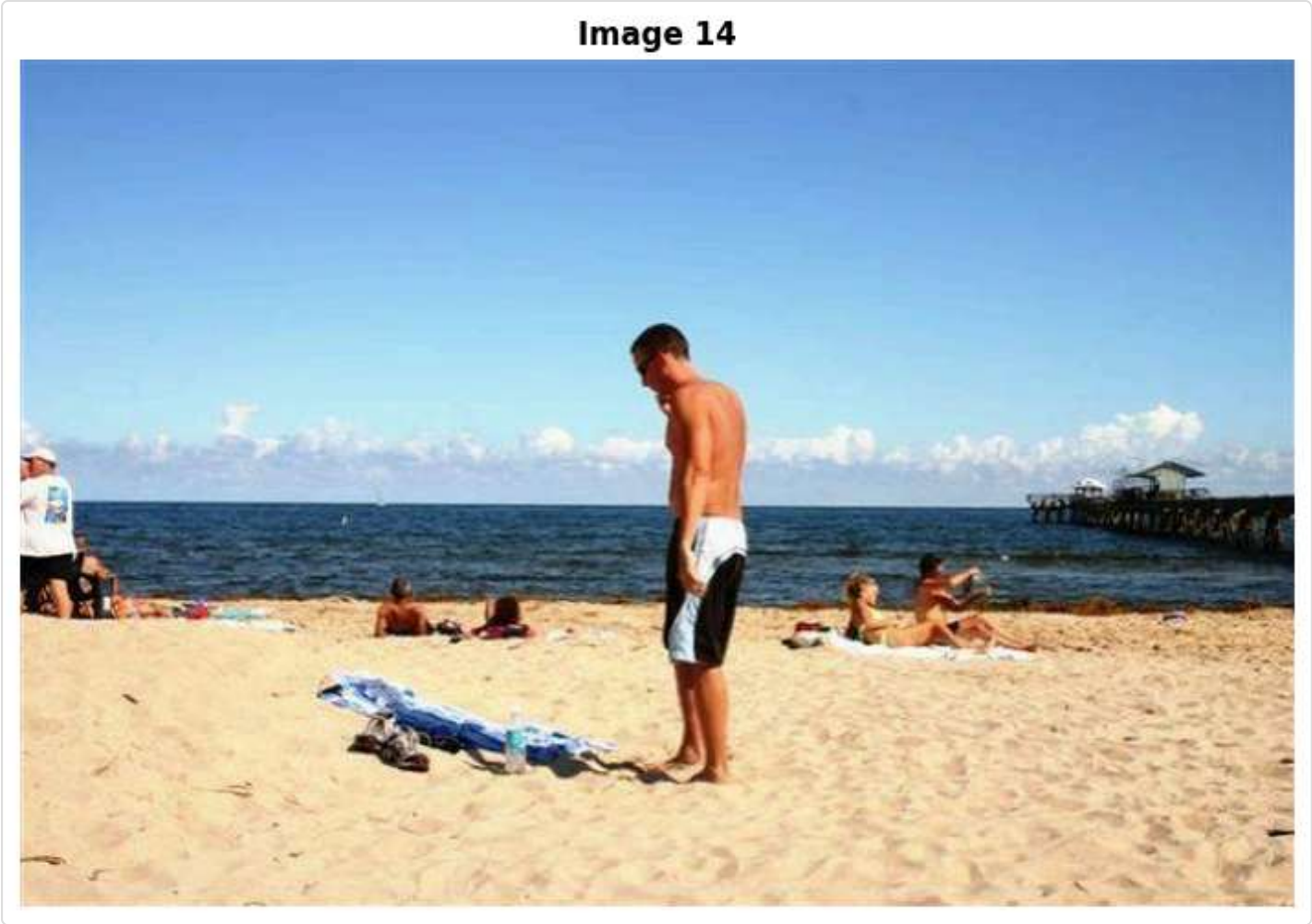


Figure 6: Failure Case 6 — Person with cell phone

Ground Truth: ['carry cell phone', 'talk on cell phone']

Prediction: ['look at towel', 'wear swim trunks']

F1 Score: 0.00

Analysis: The model completely missed the cell phone, likely due to small object size or occlusion, and focused on more salient objects instead.

Summary of Failure Reasons

Table 4: Summary of Failure Modes

Failure Type	Count	Description
Object Misidentification	3	Wrong object identified (kite→parachute, bowl→baby, missed phone)
Lexical Mismatch	2	Semantically correct but different words (kayak vs boat, chair vs table)
Dataset Annotation Issues	2	Ground truth quality problems (no interaction cases)

Failure Type	Count	Description
Action Mismatch	1	Correct object, wrong verb (blow vs cut)

↑ Back to top

Improvement Attempts

To address the failure cases, two improved prompting strategies were tested on the three worst failures.

Strategy 1: Few-Shot Learning

A prompt was created with example images and their correct HOI annotations to help the model learn the expected output format and terminology.

Strategy 2: Chain-of-Thought Reasoning

The model was asked to reason step-by-step: identify people, identify objects, then determine interactions.

Results of Improvement Attempts

Table 5: Improvement Attempt Results

Image	Original	Few-Shot	CoT	Best	Change
Image 1 (Kayak)	0.00	0.67	0.00	Few-Shot	+0.67
Image 3 (Kite)	0.00	0.80	0.00	Few-Shot	+0.80
Image 6 (Bowl)	0.00	0.00	0.00	Neither	0.00
Average	**0.00**	**0.49**	**0.00**		**+0.49**

Image 1: Kayak Scene — FIXED

Figure 7: Improvement attempt on Image 1

Results:

- **Ground Truth:** ['ride boat', 'row boat', 'sit on boat']
- **Original Prediction:** ['hold paddle', 'sit on kayak', 'wear vest'] (F1: 0.00)
- **Few-Shot Prediction:** ['hold paddle', 'ride boat', 'sit on boat'] (F1: 0.67)
- **Chain-of-Thought Prediction:** ['hold paddle', 'sit on kayak', 'wear life vest'] (F1: 0.00)
- **Best Result:** Few-Shot (Improvement: +0.67)

Few-shot prompting achieved F1 = 0.67 by aligning the model's terminology with ground truth. By providing examples using “boat” instead of “kayak,” the model learned to use the expected generic term. This demonstrates the original failure was due to vocabulary mismatch rather than visual understanding.

Image 3: Kite Flying — FIXED

Figure 8: Improvement attempt on Image 3

Results:

- **Ground Truth:** ['fly kite', 'hold kite', 'pull kite']
- **Original Prediction:** ['hold parachute', 'wear clothing'] (F1: 0.00)
- **Few-Shot Prediction:** ['fly kite', 'hold kite'] (F1: 0.80)
- **Chain-of-Thought Prediction:** ['hold paraglider', 'look at paraglider'] (F1: 0.00)
- **Best Result:** Few-Shot (Improvement: +0.80)

Few-shot prompting achieved F1 = 0.80 by correcting object misidentification. Providing visual examples helped the model distinguish between kites and parachutes.

Image 6: Bowl Scene — UNSOLVED

Figure 9: Improvement attempt on Image 6

Results:

- **Ground Truth:** ['hold bowl', 'lick bowl']
- **Original Prediction:** ['hold baby', 'wear sweater'] (F1: 0.00)
- **Few-Shot Prediction:** ['hold baby', 'wear shirt'] (F1: 0.00)
- **Chain-of-Thought Prediction:** ['hold vase'] (F1: 0.00)
- **Best Result:** Neither (Improvement: 0.00)

Neither strategy improved performance. The model consistently misidentified the bowl, indicating a fundamental visual perception problem that cannot be overcome through prompting alone.

↑ Back to top

Discussion

What Prompting Can Fix

Few-shot prompting is effective for:

- **Terminology Alignment:** When the model uses different but semantically equivalent terms
- **Ambiguous Object Recognition:** When objects have similar visual features

What Prompting Cannot Fix

Prompt engineering has clear limitations:

- **Fundamental Visual Errors:** Complete object misidentification
- **Small Object Detection:** Objects too small or occluded
- **Dataset Quality Issues:** Incorrect ground truth annotations

Evaluation Metric Limitations

F1 scores based on strict string matching do not accurately reflect model quality. Several "failures" were cases where the model provided more accurate predictions than ground truth. Better evaluation metrics are needed, such as semantic similarity measures or human evaluation.

↑ Back to top