

HW6 Report - Juntong Peng

3.1 Task 1: Single-Instance Detection

3.1.1 Running LOADnet2 on PurdueShapes5

Training loss

- Classification loss curve:

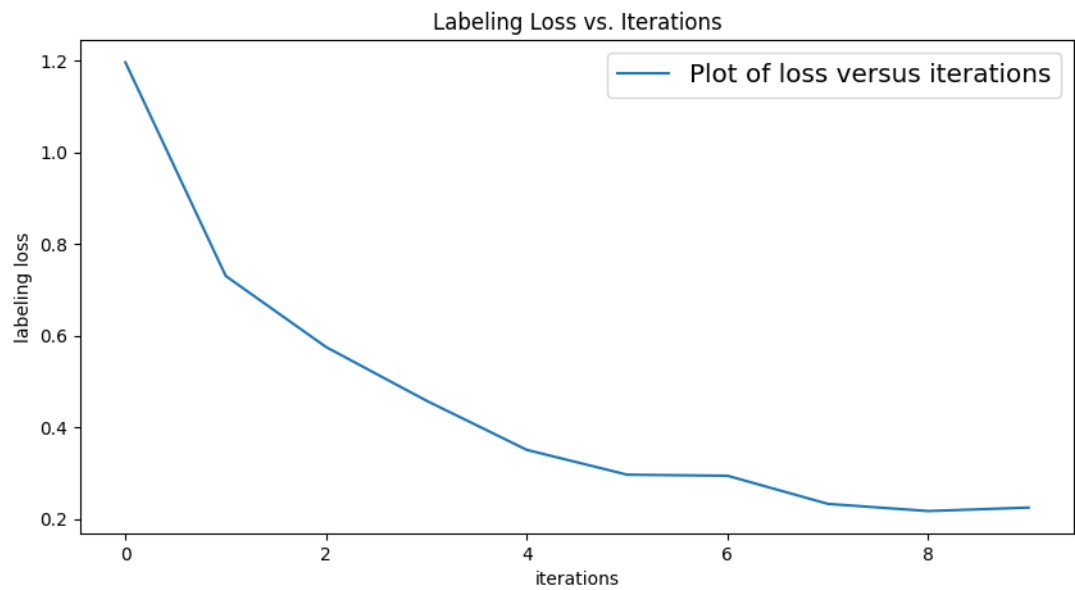


Figure 1. Task 1 classification-loss curve.

- Regression loss curve:

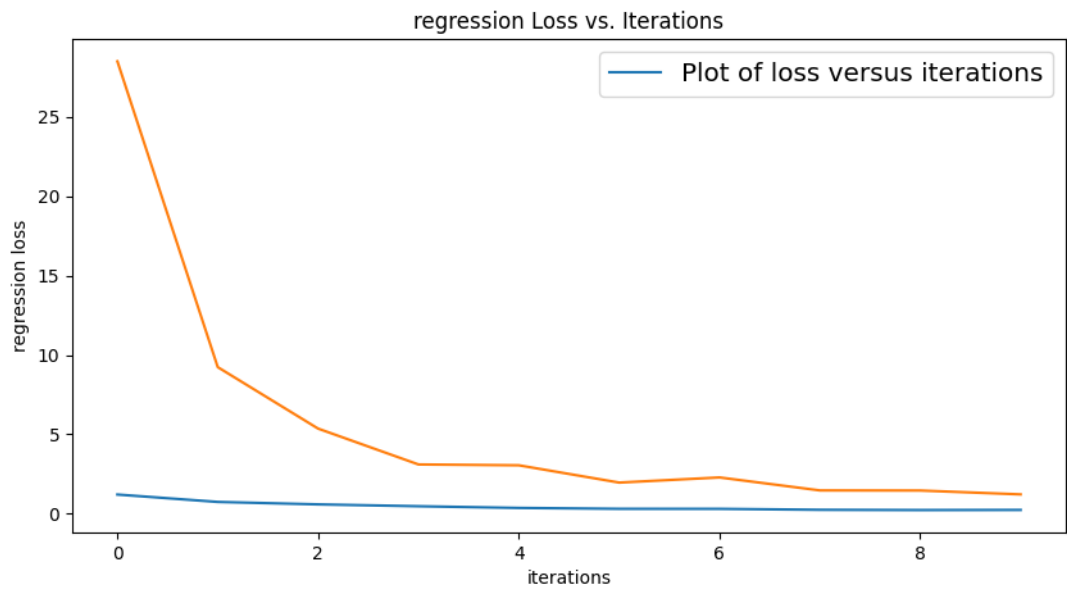


Figure 2. Task 1 regression-loss curve.

Test visualization

- Ground-truth vs. predicted bounding boxes with predicted class labels:

Task 1 Test Predictions (6 samples)

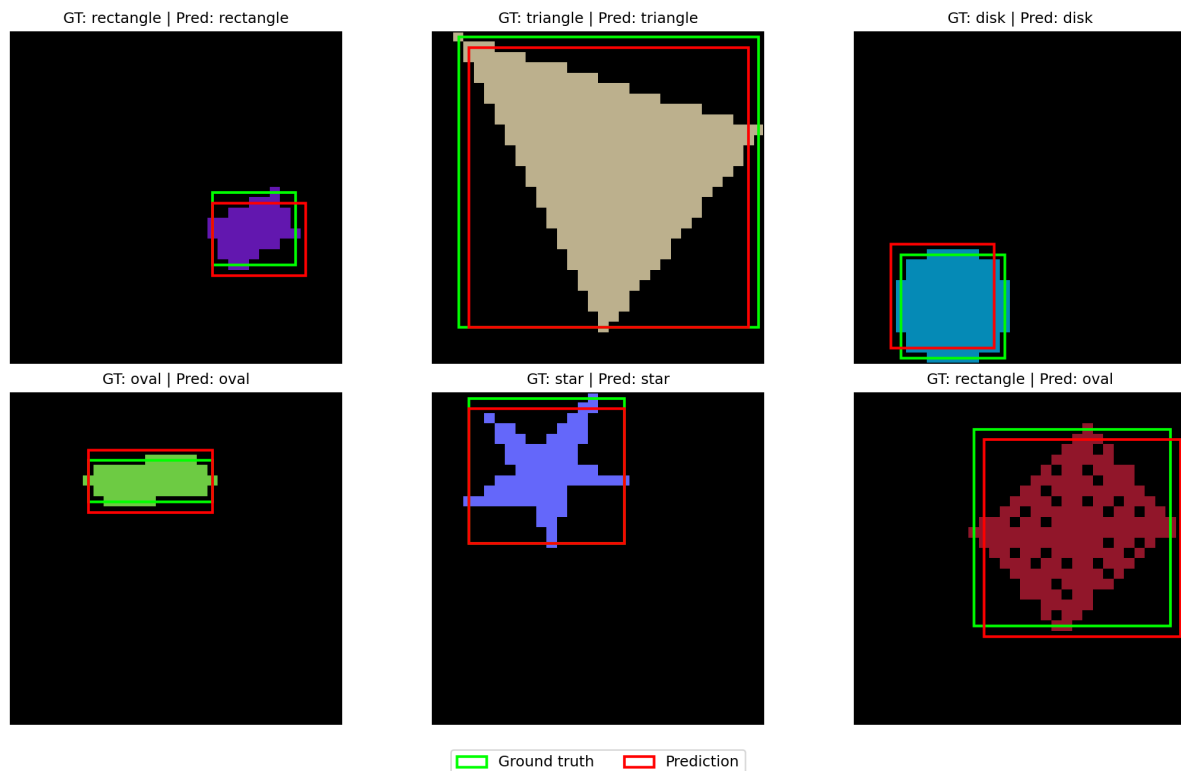


Figure 3. Task 1 test-image grid with ground-truth and predicted bounding boxes.

Classification accuracy on the test set

- Overall test accuracy: **88.9%**
- Per-class accuracy:
 - `oval`: 95.5%
 - `star`: 99.5%
 - `disk`: 97.0%
 - `triangle`: 99.0%
 - `rectangle`: 53.5%

3.1.2 Understanding Dual-Loss Training

`LOADnet2` splits after a shared `conv + maxpool` extractor into a classification head (residual blocks → 5 class logits) and a regression head (conv sequence → 4 bbox coordinates).

During training, both losses are computed from the same forward pass:

- `loss_labeling.backward(retain_graph=True)`
- `loss_regression.backward()`
- `optimizer.step()`

`retain_graph=True` prevents PyTorch from freeing the computation graph after the first `backward()`, allowing the second loss to backpropagate through the same shared layers. Both losses accumulate gradients into shared parameters before `optimizer.step()` applies a single combined update.

`nn.MSELoss` suits regression because bounding-box coordinates are continuous and we minimize geometric distance. `nn.CrossEntropyLoss` suits classification because it optimizes softmax log-likelihood over discrete class indices — neither loss is appropriate for the other task.

3.2 Task 2: Understanding YOLO Logic

3.2.1 Studying the YOLO Concepts

Image cells

YOLO divides the image into a fixed grid so prediction is spatially organized. Instead of asking one head to directly output a variable-size set of detections, each cell is responsible for objects whose center falls inside that cell. In this implementation (`image_size=128, yolo_interval=20`), the image is partitioned into a `6 × 6` grid (36 cells), and each cell contributes candidate detections through its anchors.

Anchor boxes

Each cell has multiple anchor boxes so one location can represent different object shapes/aspect ratios. In YOLOLogic there are 5 anchors per cell, with aspect-ratio bins roughly from tall-thin to wide-short. During target assignment, an object is mapped to the cell containing its center, and then to the anchor whose aspect-ratio bin matches the ground-truth box ratio. This separates "where" (cell center offset) from "shape prior" (anchor type), which stabilizes training.

yolo vector

Each (`cell, anchor`) pair is encoded as [`obj, dx, dy, bh, bw, c1, c2, c3`] (8 elements), where:

- `obj`: objectness score
- `dx, dy`: center offset from cell center, normalized by cell size
- `bh, bw`: box size relative to anchor size ($h_{GT} = \sigma_h * h_{anchor}$)
- `c1, c2, c3`: class scores (`Dr_Eval, house, watertower`)

A 9th "nil" class is appended during loss computation to represent empty slots.

yolo tensor

Shape is (`C, A, V`) per image or (`B, C, A, V`) for a batch. For this setup: `C=36, A=5, V=8` (9 during training).

3.2.2 Running Multi-Instance Detection

Training loss deliverable (all three losses)

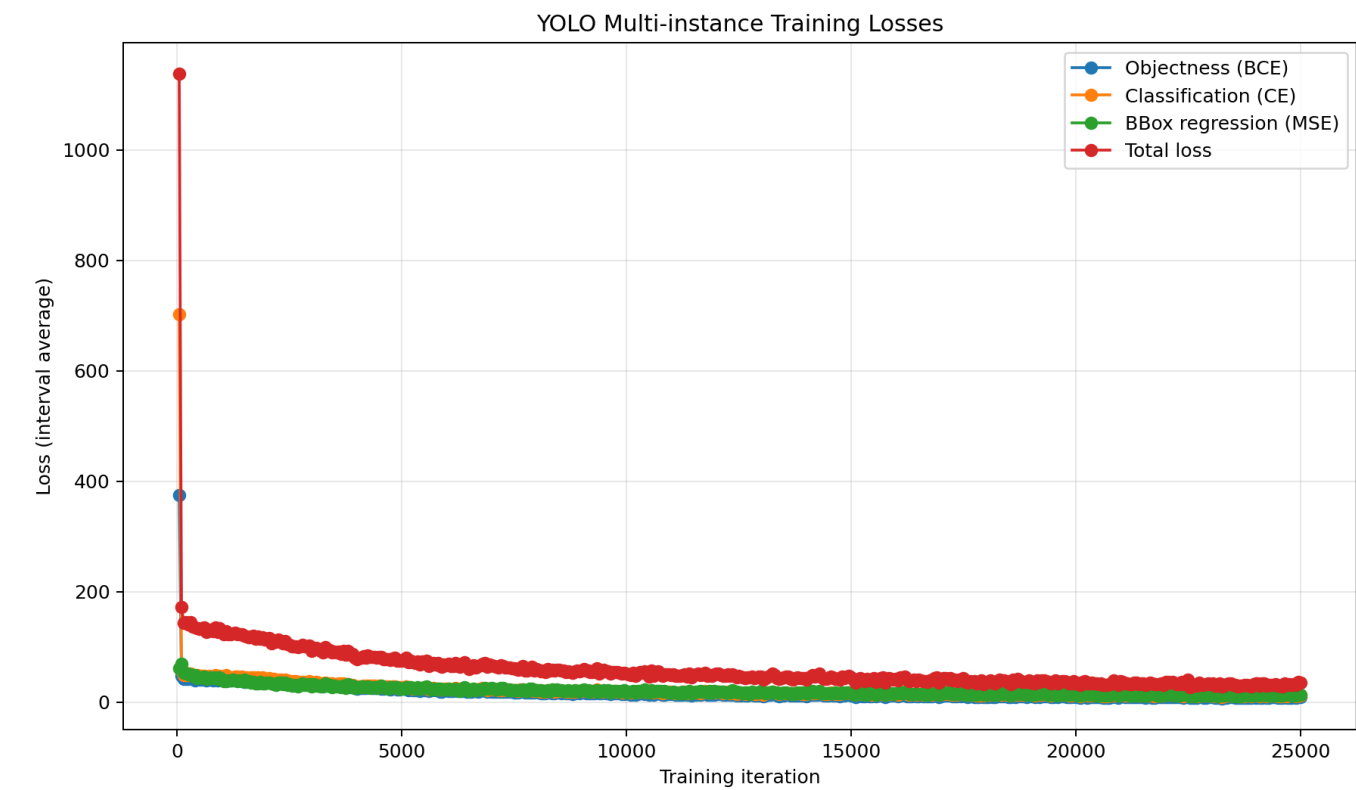


Figure 5. Task 2 objectness, classification, and regression losses (10-epoch GPU run).

Test visualization deliverable



Figure 6. Task 2 qualitative multi-instance detections with predicted and ground-truth annotations.

Quantitative results (1000 test images, IoU >= 0.5)

- Precision: 0.393, Recall: 0.440

- Per-class recall: Dr_Eval 0.609, house 0.600, watertower 0.102

Brief quality discussion

After 10 full GPU epochs, the detector achieves precision 0.393 and recall 0.440 at IoU 0.5. Dr_Eval and house classes reach ~60% recall, while watertower lags at 10% — likely due to its rarer appearance and atypical aspect ratios. The high false-positive count (1704) reflects the anchor-based decoder firing on ambiguous background cells, which would improve with longer training or NMS tuning.

3.3 Task 3: LVIS Detection Dataset

3.3.1 Detection Dataset Construction

Code snippet for detection dataset construction:

```
for ann in annotations:
    cid = ann["category_id"]
    if cid not in selected_id_set:
        continue
    if ann.get("area", 0.0) >= args.min_area:
        qual_anns_by_img[ann["image_id"]].append(ann)

eligible_img_ids = [img_id for img_id, anns in qual_anns_by_img.items() if
len(anns) > 0]
random.shuffle(eligible_img_ids)
val_ids = set(eligible_img_ids[:num_val])
train_ids = set(eligible_img_ids[num_val:])

for ann in qual_anns_by_img[img_id]:
    x, y, w, h = ann["bbox"]
    scaled = clamp_bbox_xywh(x * scale_x, y * scale_y, w * scale_x, h *
scale_y, args.target_size)
    objects.append({"category_index": cat_idx, "bbox": [round(v, 2) for v
in scaled]})

ann_payload = {
    "image_id": img_id,
    "file_name": file_name,
    "objects": [{"category_index": o["category_index"], "bbox": o["bbox"]}
for o in objects],
}
out_ann_path.write_text(json.dumps(ann_payload, indent=2))
```

with the following selected LVIS categories:

- person (LVIS ID 793)
- horse (LVIS ID 569)
- elephant (LVIS ID 422)

These satisfy Section 2.6 constraints (≥ 500 images/category and multi-instance ratio $\geq 30\%$).

Section 2.6 verification output

```
HW6 Section 2.6 Verification Output
Selected categories: ['person', 'horse', 'elephant']
Raw image count per category:
  person: 1928
  horse: 1904
  elephant: 1878
Multi-instance ratio per category:
  person: 0.3791
  horse: 0.3577
  elephant: 0.4840
Train images: 3925
Validation images: 981
Training object instances per category:
  person: 2648
  horse: 2152
  elephant: 2909
Average foreground objects per image (train): 1.9641
Average foreground objects per image (overall): 1.9566
```

3x3 sample image grid with boxes/labels (3 images per category)

HW6 Task 3.3.1: 3x3 LVIS Detection Samples

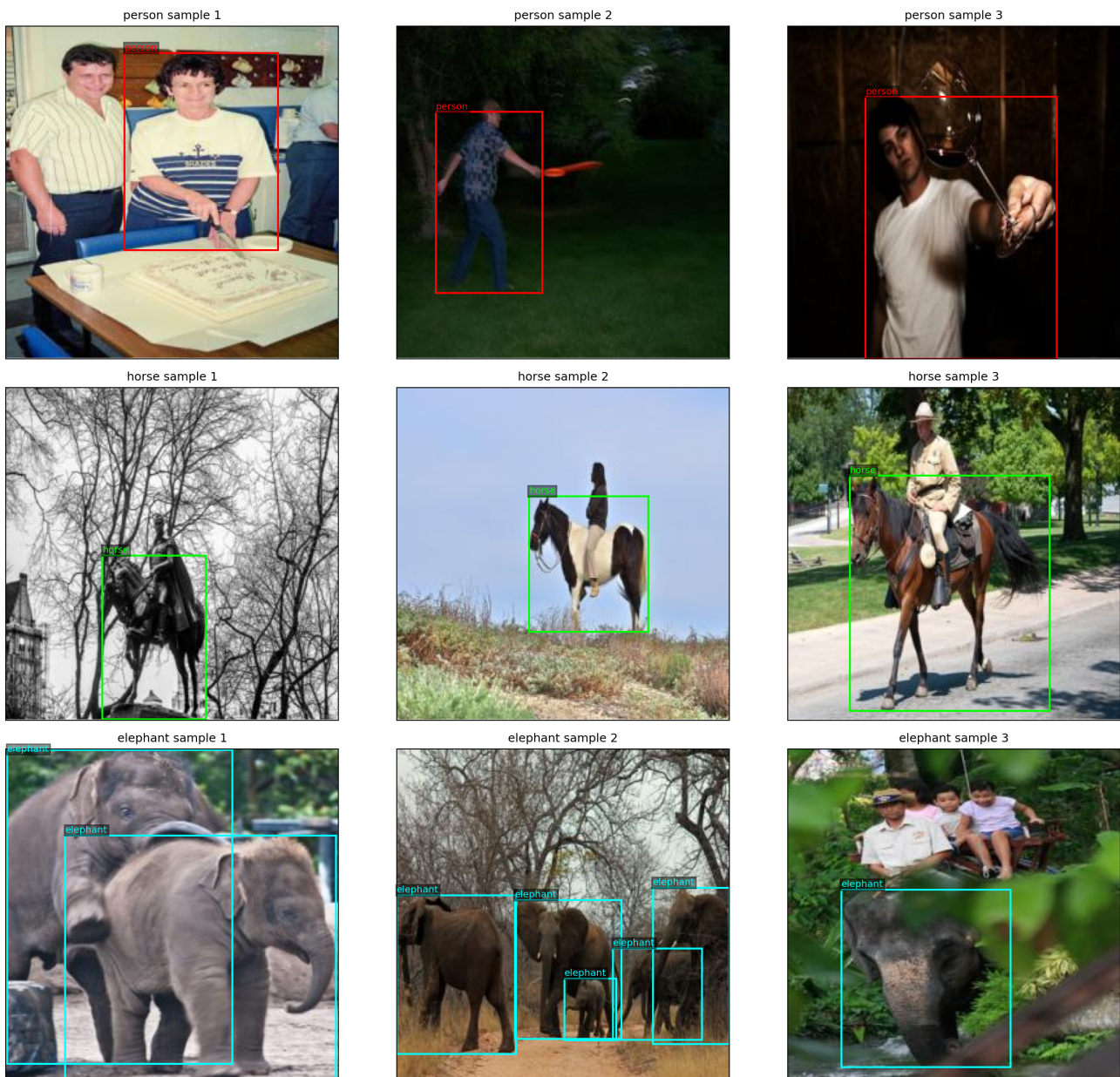


Figure 7. 3x3 sample grid from the resized LVIS detection dataset with bounding boxes and labels.

3.3.2 Segmentation Mask Construction

Code snippet for segmentation mask construction:

```
split_manifest = json.loads((args.detection_root / "metadata" /
"split_manifest.json").read_text())
all_split_ids = set(split_manifest["train_image_ids"]) |
set(split_manifest["val_image_ids"])

for ann in lvis["annotations"]:
    if ann["image_id"] not in all_split_ids:
        continue
    if ann["category_id"] not in selected_set or ann.get("area", 0.0) <
args.min_area:
```

```
        continue
    anns_by_img[ann["image_id"]].append(ann)

mask_orig = np.zeros((h_orig, w_orig), dtype=np.uint8)
for ann in anns_by_img.get(image_id, []):
    binary = segmentation_to_binary_mask(ann.get("segmentation"), h_orig,
w_orig)
    if binary is not None:
        mask_orig[binary] = lvis_id_to_mask_idx[ann["category_id"]]

mask_resized = Image.fromarray(mask_orig,
mode="L").resize((args.target_size, args.target_size), Image.NEAREST)
mask_resized.save(out_mask_path)
```

Critical same-split requirement check

- Same train images as detection dataset: `True`
- Same val images as detection dataset: `True`
- Train masks written: `3925` (matches train image count)
- Val masks written: `981` (matches val image count)

Mask format

- Single-channel PNG masks with values in `{0,1,2,3}`
 - `0`: background
 - `1`: person
 - `2`: horse
 - `3`: elephant
- Resizing uses nearest-neighbor interpolation (`Image.NEAREST`) to preserve class IDs.

6 sample image/mask visualizations (2 per category)

Task 3.3.2 Samples: Left=Image, Right=Segmentation Mask

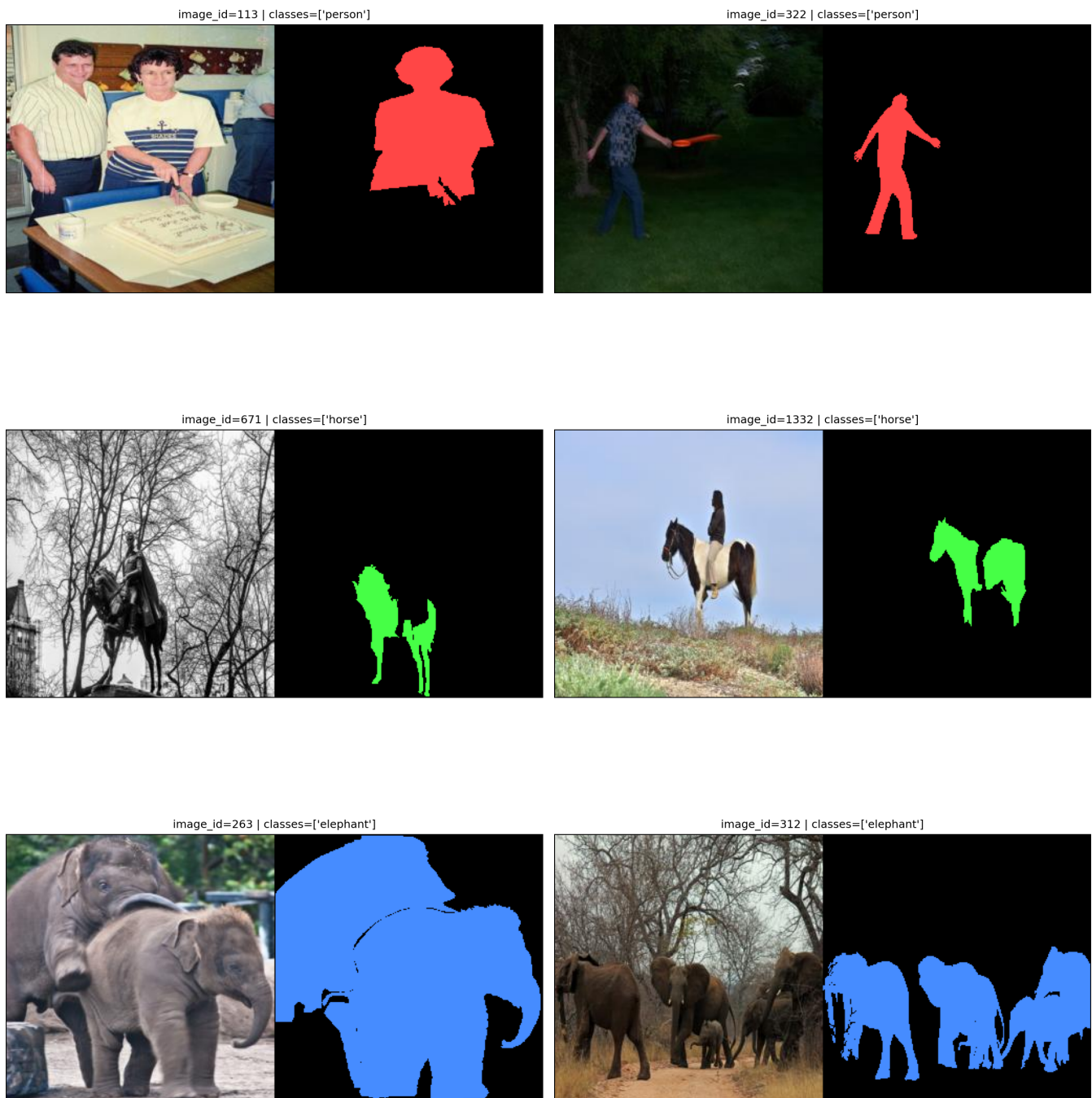


Figure 8. Task 3.3.2 sample image/mask pairs, with two examples per foreground category.

Per-class pixel distribution (training set, including background)

- background (0):0.801266
- person (1):0.058241
- horse (2):0.047818

- elephant (3):0.092675

3.3.3 IoU Computation

Complete implementation code:

```
def compute_iou(box1, box2):
    x1, y1, w1, h1 = [float(v) for v in box1]
    x2, y2, w2, h2 = [float(v) for v in box2]
    b1_x1, b1_y1, b1_x2, b1_y2 = x1, y1, x1 + max(0.0, w1), y1 + max(0.0,
h1)
    b2_x1, b2_y1, b2_x2, b2_y2 = x2, y2, x2 + max(0.0, w2), y2 + max(0.0,
h2)

    inter_x1 = max(b1_x1, b2_x1)
    inter_y1 = max(b1_y1, b2_y1)
    inter_x2 = min(b1_x2, b2_x2)
    inter_y2 = min(b1_y2, b2_y2)
    inter_w = max(0.0, inter_x2 - inter_x1)
    inter_h = max(0.0, inter_y2 - inter_y1)
    inter_area = inter_w * inter_h

    area1 = max(0.0, w1) * max(0.0, h1)
    area2 = max(0.0, w2) * max(0.0, h2)
    union_area = area1 + area2 - inter_area
    return 0.0 if union_area <= 0.0 else float(inter_area / union_area)

def compute_iou_matrix(boxes1, boxes2):
    b1_xy = boxes1[:, :2]
    b1_wh = boxes1[:, 2:].clamp(min=0)
    b2_xy = boxes2[:, :2]
    b2_wh = boxes2[:, 2:].clamp(min=0)

    b1_x1y1 = b1_xy
    b1_x2y2 = b1_xy + b1_wh
    b2_x1y1 = b2_xy
    b2_x2y2 = b2_xy + b2_wh

    inter_x1y1 = torch.maximum(b1_x1y1[:, None, :], b2_x1y1[None, :, :])
    inter_x2y2 = torch.minimum(b1_x2y2[:, None, :], b2_x2y2[None, :, :])
    inter_wh = (inter_x2y2 - inter_x1y1).clamp(min=0)
    inter_area = inter_wh[..., 0] * inter_wh[..., 1]

    area1 = b1_wh[:, 0] * b1_wh[:, 1]
    area2 = b2_wh[:, 0] * b2_wh[:, 1]
    union_area = area1[:, None] + area2[None, :] - inter_area

    iou = torch.zeros_like(inter_area, dtype=torch.float32)
    valid = union_area > 0
    iou[valid] = inter_area[valid] / union_area[valid]
    return iou
```


Required test cases:

- (a) Non-overlapping boxes: $\text{IoU} = 0.000000$
- (b) Identical boxes: $\text{IoU} = 1.000000$
- (c) Partially overlapping boxes: $\text{IoU} = 0.142857$ (25/175)

3.4 Task 4: YOLO Network for LVIS

3.4.1 Designing YOLO Parameters

Chosen YOLO parameters for the 256×256 LVIS dataset:

- Grid size: 8×8 (`grid_size = 8`)
- Number of cells: $C = 64$
- Anchor boxes per cell: $A = 5$
- Anchor aspect ratios (h/w): $[1/5, 1/3, 1, 3, 5]$
- Number of classes: $K = 3$ (`person, horse, elephant`)
- YOLO vector length: $V = 1 + K + 4 = 8$

Resulting yolo tensor shape:

- Unflattened: $(B, C, A, V) = (B, 64, 5, 8)$
- Flattened: $(B, C * A * V) = (B, 2560)$

YOLO vector organization used in this implementation:

- `[..., 0]`: objectness
- `[..., 1:4]`: one-hot class (`person, horse, elephant`)
- `[..., 4:8]`: `[delta_x, delta_y, sigma_w, sigma_h]`

Clear tensor-organization description:

- For each image, index first by cell ($0..63$), then anchor ($0..4$), then vector slot ($0..7$).
- Example access pattern: `yolo_tensor[cell_idx, anchor_idx, :]` gives one prediction target vector for that cell-anchor pair.

Measured verification (from `results/hw6_task4/task4_1_3_summary.json`):

- Dummy input shape $[2, 3, 256, 256]$ -> model output shape $[2, 2560]$

3.4.2 Network Architecture: LVISYOLONet

Complete `LVISYOLONet` code:

```
class SkipBlock(nn.Module):
    """Residual block with optional downsampling and channel projection."""
    def __init__(self, in_channels: int, out_channels: int, downsample:
bool = False) -> None:
        super().__init__()
        stride = 2 if downsample else 1
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
```

```

stride=stride, padding=1, bias=False)
    self.bn1 = nn.BatchNorm2d(out_channels)
    self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
stride=1, padding=1, bias=False)
    self.bn2 = nn.BatchNorm2d(out_channels)
    needs_proj = downsample or (in_channels != out_channels)
    self.proj = (
        nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=1,
stride=stride, bias=False),
            nn.BatchNorm2d(out_channels),
        )
        if needs_proj
        else nn.Identity()
    )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        identity = self.proj(x)
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out = F.relu(out + identity)
        return out

class LVISYOLONet(nn.Module):
    """YOLO-style detector backbone/head for 256x256 LVIS images."""
    def __init__(self, num_classes: int = 3, num_cells: int = 64,
num_anchors: int = 5) -> None:
        super().__init__()
        grid_size = int(num_cells**0.5)
        if grid_size * grid_size != num_cells:
            raise ValueError("num_cells must be a perfect square (e.g., 16
or 64).")

        self.num_classes = num_classes
        self.num_anchors = num_anchors
        self.num_cells = num_cells
        self.grid_size = grid_size
        self.yolo_vec_len = 1 + num_classes + 4
        self.out_size = num_cells * num_anchors * self.yolo_vec_len

        self.stem_conv = nn.Conv2d(3, 64, kernel_size=3, padding=1,
bias=False)
        self.stem_bn = nn.BatchNorm2d(64)
        self.stage1 = nn.Sequential(SkipBlock(64, 64), SkipBlock(64, 64))
        self.down1 = SkipBlock(64, 128, downsample=True)
        self.stage2 = nn.Sequential(SkipBlock(128, 128), SkipBlock(128,
128))
        self.down2 = SkipBlock(128, 256, downsample=True)
        self.stage3 = nn.Sequential(SkipBlock(256, 256), SkipBlock(256,
256))
        self.down3 = SkipBlock(256, 256, downsample=True)
        self.stage4 = nn.Sequential(SkipBlock(256, 256), SkipBlock(256,
256))

```



```

        self.head_pool = nn.AdaptiveAvgPool2d((4, 4))
        self.fc1 = nn.Linear(256 * 4 * 4, 2048)
        self.fc2 = nn.Linear(2048, self.out_size)
        self.dropout = nn.Dropout(p=0.2)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = F.relu(self.stem_bn(self.stem_conv(x)))
        x = F.max_pool2d(x, kernel_size=2, stride=2)
        x = self.stage1(x)
        x = self.down1(x)
        x = self.stage2(x)
        x = self.down2(x)
        x = self.stage3(x)
        x = self.down3(x)
        x = self.stage4(x)
        x = self.head_pool(x)
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x

```

Total learnable parameters:

- 21,495,360

3.4.3 Custom Dataloader

Complete dataloader code (core logic):

```

def assign_cell_and_anchor(bbox, img_size, grid_size, anchor_ratios):
    x, y, w, h = [float(v) for v in bbox]
    cx = x + w / 2.0
    cy = y + h / 2.0
    cell_size = img_size / float(grid_size)
    col = int(cx / cell_size)
    row = int(cy / cell_size)
    col = max(0, min(col, grid_size - 1))
    row = max(0, min(row, grid_size - 1))
    cell_idx = row * grid_size + col

    cell_cx = (col + 0.5) * cell_size
    cell_cy = (row + 0.5) * cell_size
    delta_x = (cx - cell_cx) / cell_size
    delta_y = (cy - cell_cy) / cell_size

    anchor_area = cell_size * cell_size
    best_anchor, best_iou = 0, -1.0
    best_a_w, best_a_h = cell_size, cell_size
    for anchor_idx, ratio in enumerate(anchor_ratios):
        a_h = math.sqrt(anchor_area * ratio)
        a_w = a_h / ratio

```

```

        anchor_box = [cell_cx - a_w / 2.0, cell_cy - a_h / 2.0, a_w, a_h]
        iou = compute_iou(bbox, anchor_box)
        if iou > best_iou:
            best_iou = iou
            best_anchor = anchor_idx
            best_a_w, best_a_h = a_w, a_h

    sigma_w = w / best_a_w if best_a_w > 0 else 1.0
    sigma_h = h / best_a_h if best_a_h > 0 else 1.0
    return cell_idx, best_anchor, [delta_x, delta_y, sigma_w, sigma_h]

def build_gt_yolo_tensor(anns, img_size, grid_size, anchor_ratios,
num_classes):
    num_cells = grid_size * grid_size
    num_anchors = len(anchor_ratios)
    vec_len = 1 + num_classes + 4
    gt = torch.zeros(num_cells, num_anchors, vec_len, dtype=torch.float32)
    for ann in anns:
        cls_idx = int(ann["category_idx"])
        cell_idx, anchor_idx, offsets = assign_cell_and_anchor(
            ann["bbox"], img_size, grid_size, anchor_ratios
        )
        gt[cell_idx, anchor_idx, 0] = 1.0
        gt[cell_idx, anchor_idx, 1 + cls_idx] = 1.0
        gt[cell_idx, anchor_idx, 1 + num_classes:] = torch.tensor(offsets,
dtype=torch.float32)
    return gt

class LVISYoloDataset(Dataset):
    def __getitem__(self, idx):
        image_path = self.image_paths[idx]
        ann_path = self.ann_dir / f"{image_path.stem}.json"
        image = Image.open(image_path).convert("RGB")
        image_tensor = self.transform(image)

        ann_payload = json.loads(ann_path.read_text())
        anns = [
            {"category_idx": int(obj["category_index"]), "bbox": [float(v)
for v in obj["bbox"]]}
            for obj in ann_payload["objects"]
        ]
        gt_yolo_tensor = build_gt_yolo_tensor(
            anns=anns,
            img_size=self.img_size,
            grid_size=self.grid_size,
            anchor_ratios=self.anchor_ratios,
            num_classes=self.num_classes,
        )
        gt_yolo_flat = gt_yolo_tensor.reshape(-1)
        object_targets = []
        for ann in anns:
            cell_idx, anchor_idx, offsets = assign_cell_and_anchor(
                bbox=ann["bbox"],
                img_size=self.img_size,

```

```

        grid_size=self.grid_size,
        anchor_ratios=self.anchor_ratios,
    )
    object_targets.append(
        {"cell_idx": cell_idx, "anchor_idx": anchor_idx,
"category_idx": ann["category_idx"]}
    )
    return {
        "image": image_tensor,
        "gt_yolo_tensor": gt_yolo_tensor,
        "gt_yolo_flat": gt_yolo_flat,
        "object_targets": object_targets,
        "file_name": image_path.name,
        "image_id": int(ann_payload["image_id"]),
    }

```

- Train dataset length: 3925
- Val dataset length: 981
- Batch shapes:
 - images: [4, 3, 256, 256]
 - gt_yolo_tensor: [4, 64, 5, 8]
 - gt_yolo_flat: [4, 2560]

3.4.4 Training and Evaluation

Trained **LVISYOLONet** with the required three-loss setup using:

- `python -u run_hw6_task4_4_train_eval.py --epochs 15 --batch-size 16 --lr 1e-4 --grid-size 8 --img-size 256 --conf-thresh 0.35 --nms-iou-thresh 0.5 --match-iou-thresh 0.5 --num-workers 0 --log-every 20 --val-log-every 100 2>&1 | tee results/hw6_task4/task3_4_4_train_log.txt`

Training configuration matches the spec:

- Optimizer: `Adam(lr=1e-4)`
- Epochs: 15
- Batch size: 16
- Image size: 256 x 256

Three-loss training loop (code excerpt from `run_hw6_task4_4_train_eval.py`):

```

obj_loss = bce(obj_logits, obj_target)
pos_mask = obj_target > 0.5
if pos_mask.any():
    cls_logits = pred[..., 1 : 1 + num_classes][pos_mask]
    cls_target = torch.argmax(gt[..., 1 : 1 + num_classes], dim=-1)
[pos_mask]
    cls_loss = ce(cls_logits, cls_target)
    bbox_pred = pred[..., 1 + num_classes :][pos_mask]
    bbox_target = gt[..., 1 + num_classes :][pos_mask]

```

```
bbox_loss = mse(bbox_pred, bbox_target)
else:
    cls_loss = pred.sum() * 0.0
    bbox_loss = pred.sum() * 0.0
total_loss = obj_loss + cls_loss + bbox_loss
```

Deliverable: all three loss curves

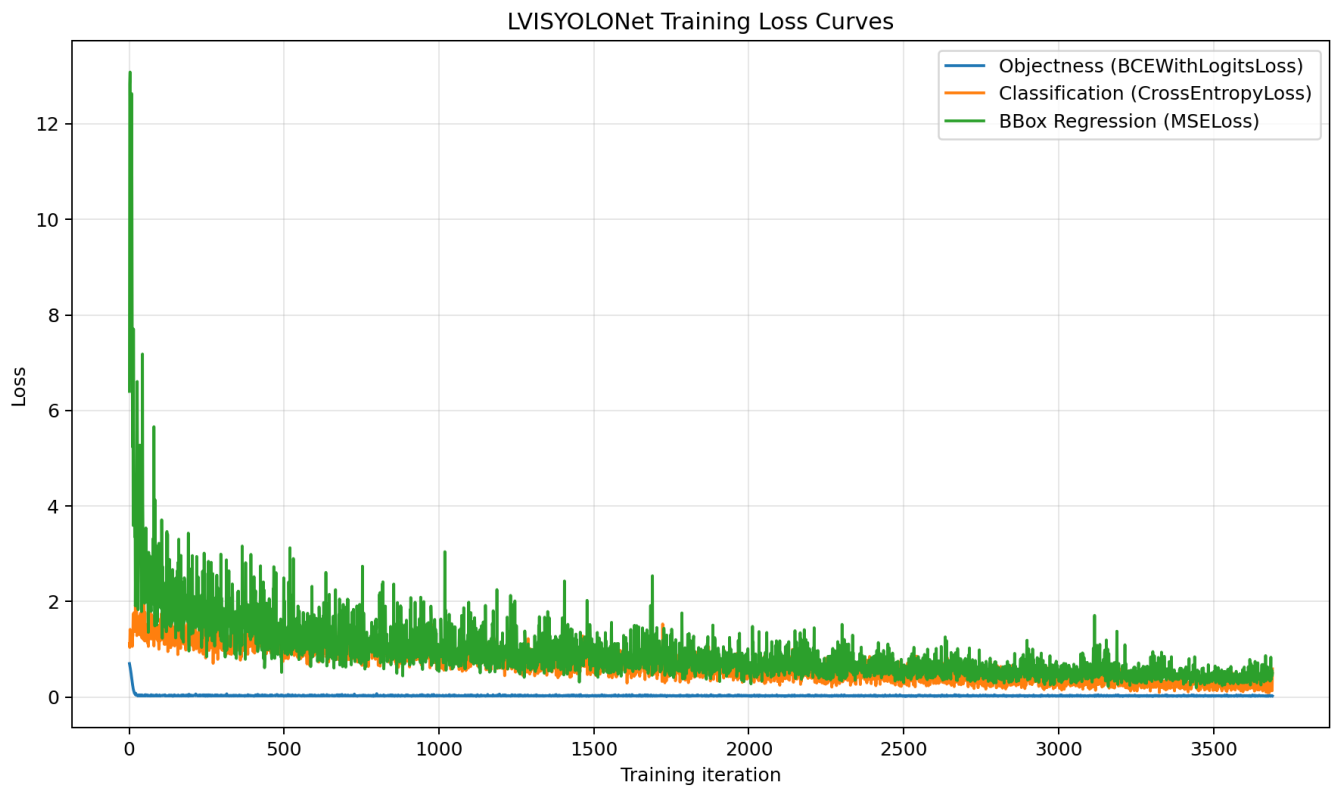


Figure 9. Task 3.4.4 LVISYOLONet objectness, classification, and regression loss curves.

Deliverable: 12 validation images (8 good + 4 failure)

Task 3.4.4 Validation Predictions (Green=GT, Red=Pred)

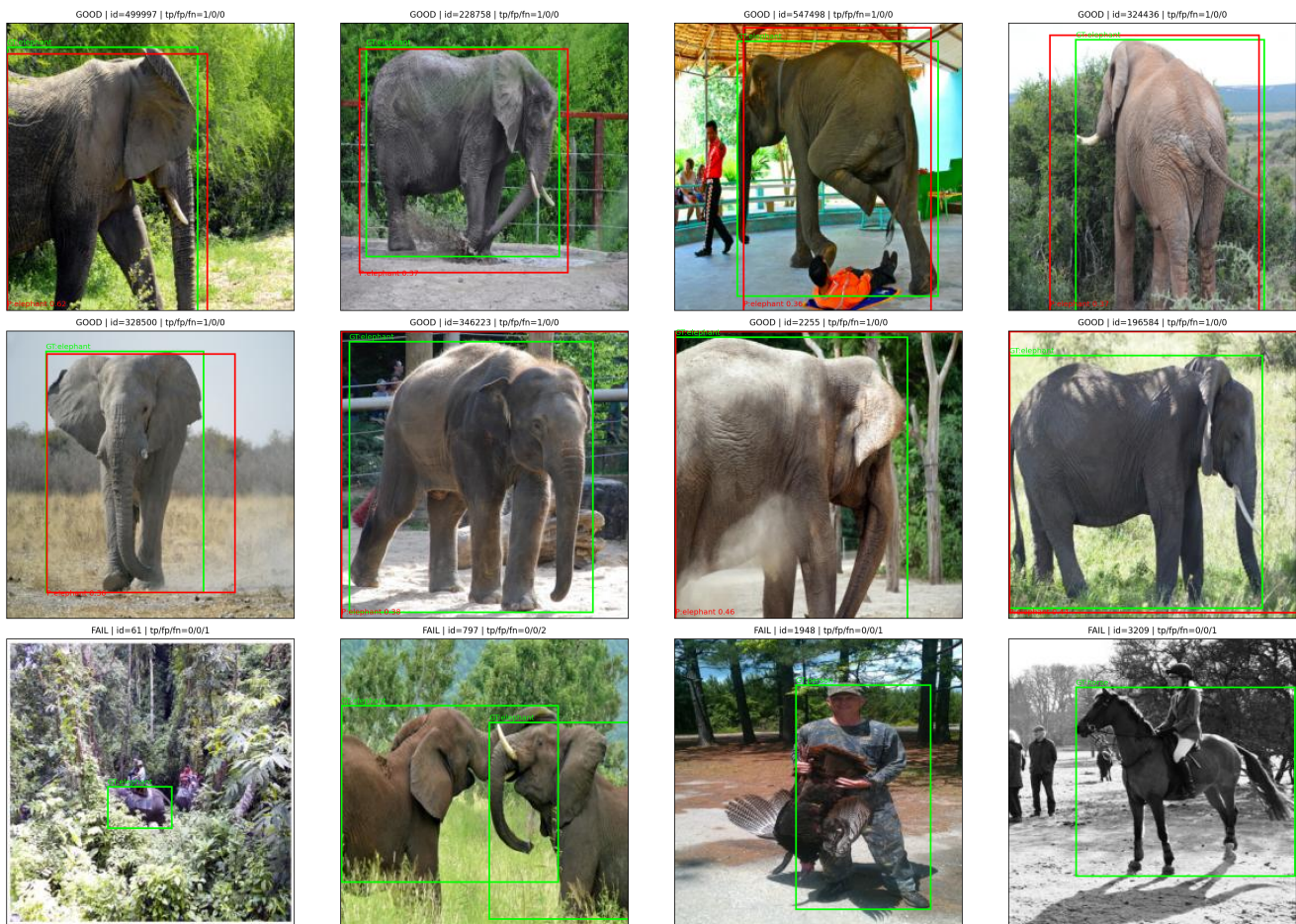


Figure 10. Task 3.4.4 qualitative validation detections with 8 good and 4 failure cases.

Validation quantitative summary (981 val images evaluated):

- Overall: precision 0.8246, recall 0.0249, F1 0.0483 (TP=47, FP=10, FN=1843)
- Per-class recall:
 - person: 0.0000
 - horse: 0.0036
 - elephant: 0.0681

Performance discussion: The detector has high precision but very low recall, meaning it fires conservatively and misses many objects. It works better on some larger, clearer objects (especially elephant) and fails frequently on crowded/small or overlapping instances, with person being the hardest category in this run. This indicates under-detection and suggests that longer training, stronger backbone capacity, and improved decoding/loss balancing are needed to improve recall.

3.5 Task 5: Semantic Segmentation

3.5.1 Understanding Transpose Convolutions and mUnet Warmup

I used the official DLStudio semantic-segmentation pipeline (`DLStudio.SemanticSegmentation.mUNet` and `PurdueShapes5MultiObjectDataset`) via:

- `run_hw6_task5_1_munet.py`

This wrapper keeps the official model/dataset logic but saves report-ready figures and summaries non-interactively.

Code snippet used to launch the mUNet warmup:

```
dls = DLStudio(
    dataroot="./data/PurdueShapes5MultiObject/",
    image_size=[64, 64],
    path_saved_model="./saved_model",
    momentum=0.9,
    learning_rate=1e-4,
    epochs=6,
    batch_size=4,
    classes=("rectangle", "triangle", "disk", "oval", "star"),
    use_gpu=True,
)
segmenter = DLStudio.SemanticSegmentation(dl_studio=dls, max_num_objects=5)
dataserver_train =
DLStudio.SemanticSegmentation.PurdueShapes5MultiObjectDataset(
    train_or_test="train", dl_studio=dls, segmenter=segmenter,
    dataset_file="PurdueShapes5MultiObject-10000-train.gz"
)
dataserver_test =
DLStudio.SemanticSegmentation.PurdueShapes5MultiObjectDataset(
    train_or_test="test", dl_studio=dls, segmenter=segmenter,
    dataset_file="PurdueShapes5MultiObject-1000-test.gz"
)
segmenter.load_PurdueShapes5MultiObject_dataset(dataserver_train,
dataserver_test)
model = segmenter.mUNet(skip_connections=True, depth=16)
segmenter.run_code_for_training_for_semantic_segmentation(model)
segmenter.run_code_for_testing_semantic_segmentation(model)
```

Q1.

A 2D convolution is a linear map $y = Kx$, where K is a sparse (Toeplitz) matrix of kernel weights. Transpose convolution applies K^T : $\hat{x} = K^T y$, reversing the spatial compression of the forward pass and thus increasing resolution.

Q2.

$$H_{\text{out}} = (4 - 1) * 2 - 2 * 1 + 3 = 6 - 2 + 3 = 7$$

So the output is 7×7 .

Q3.

- `nn.ConvTranspose2d` performs learned upsampling in one operation using learnable kernels.
- `nn.Upsample` + `nn.Conv2d` separates resizing and filtering:
 - `nn.Upsample` (nearest/bilinear) is usually fixed (no learnable parameters),
 - `nn.Conv2d` after upsampling has learnable parameters.

So both approaches can include learnable parameters, but in `Upsample + Conv2d` the upsampling step itself is not learned.

Q4.

From DLStudio's `mUNet (depth=16)` implementation for `64 x 64` inputs:

- Downsampling stages: 3
 - initial max-pool after `conv_in`,
 - one downsampling in `skip64dsDN`,
 - one downsampling in `skip128dsDN`.
- Upsampling stages: 3
 - one upsampling in `skip128usUP`,
 - one upsampling in `skip64usUP`,
 - final transpose-convolution output layer `conv_out` (upsampling to full resolution).
- Encoder-decoder skip connections: saved encoder feature channels are injected into decoder features at matching scales to restore spatial detail and boundaries.

Difference from ResNet skip connections:

- mUNet encoder-decoder skips bridge *different network depths/scales* (semantic + localization fusion for reconstruction).
- ResNet skips are mostly local residual identity paths within same-stage processing blocks to ease optimization/gradient flow and preserve feature refinement, not primarily for decoder reconstruction.

mUNet training loss curve

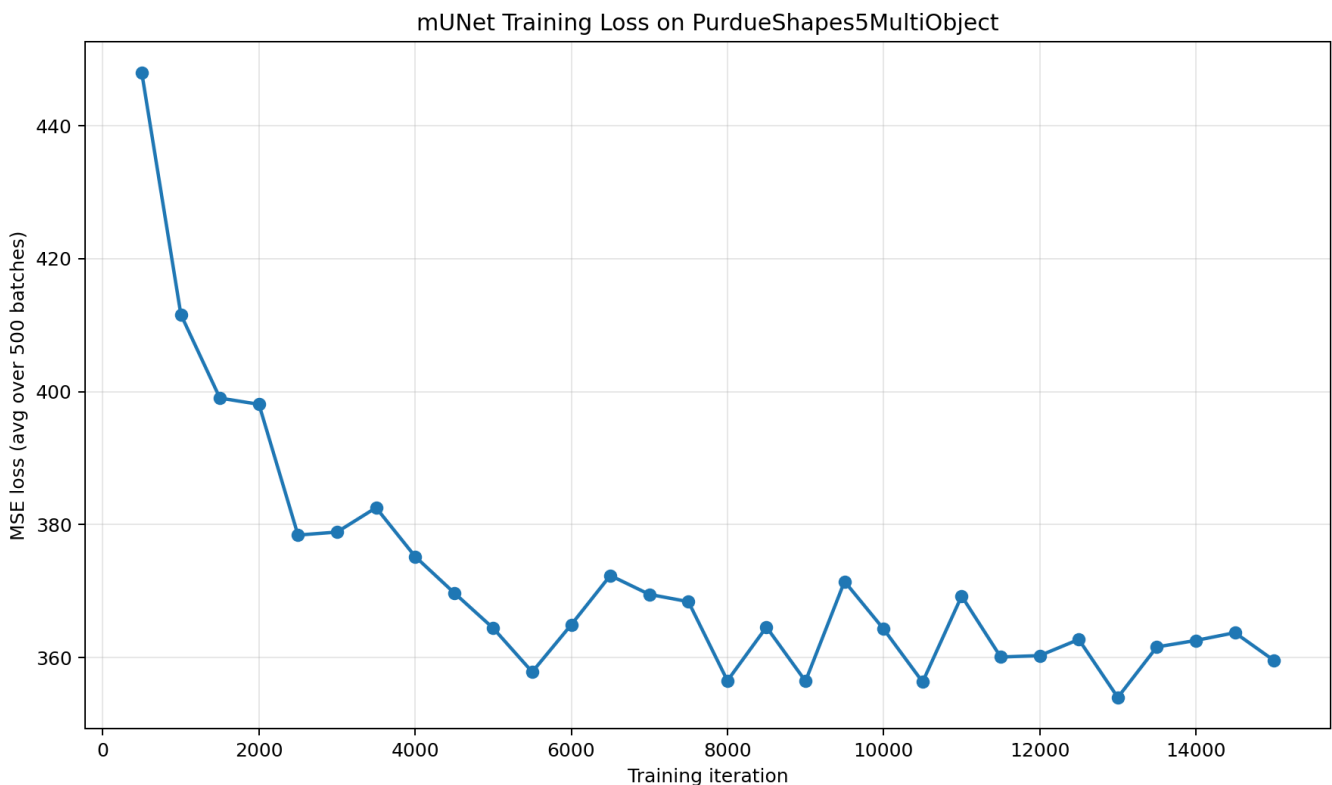


Figure 11. Task 5.1 mUNet training loss curve on PurdueShapes5MultiObject (6-epoch GPU run).

4 test images (input / GT mask / predicted mask)

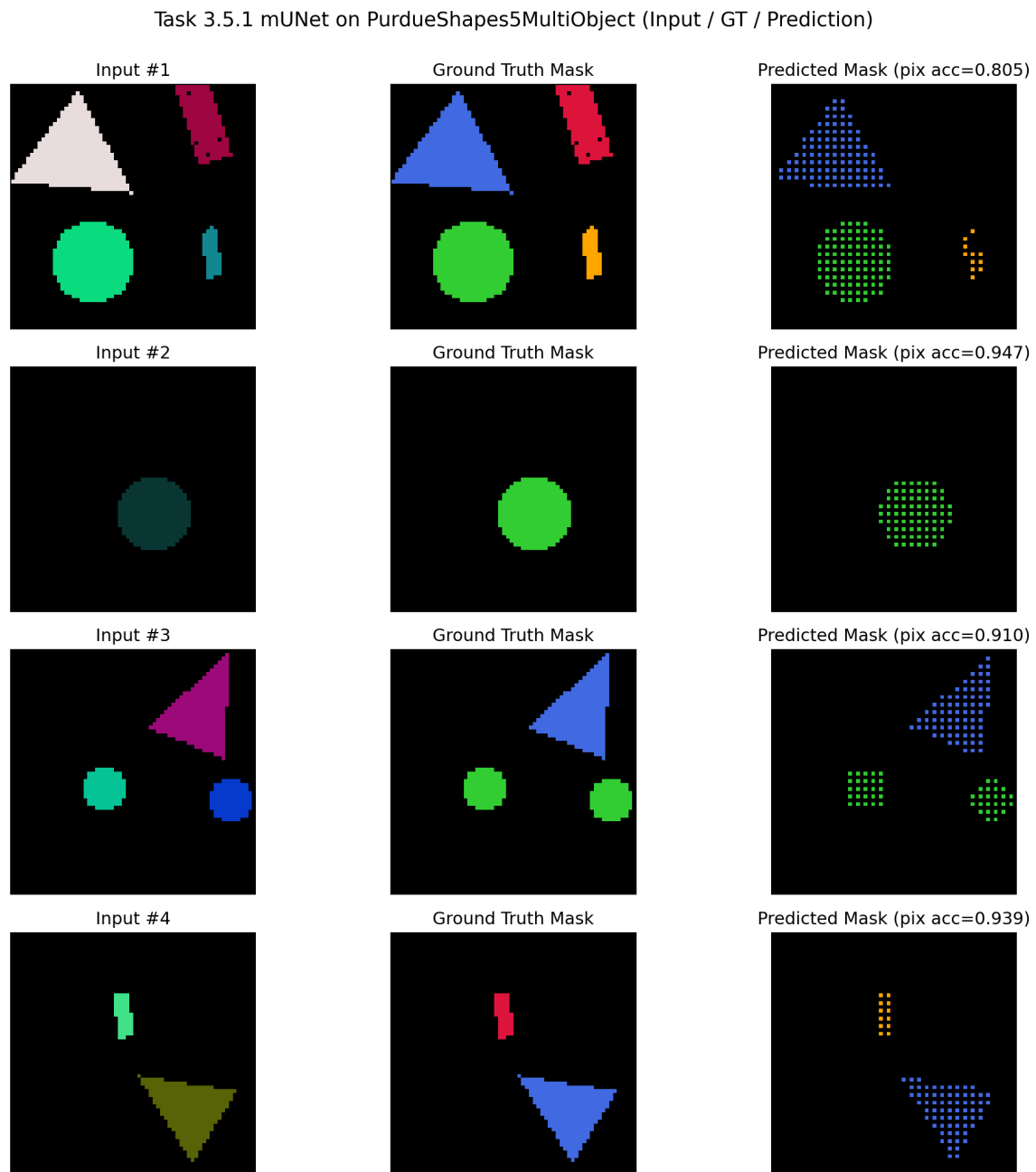


Figure 12. Task 5.1 mUNet test triplets showing input image, ground-truth mask, and predicted mask.

Brief performance paragraph: The 6-epoch GPU mUNet run on PurdueShapes5MultiObject achieved pixel accuracy **0.900** and foreground mIoU **0.238** on 4 test images. Boundaries and overlaps remain imperfect, especially for small objects and cluttered regions, but the predictions capture foreground structure and object-type regions well enough to validate the encoder-decoder pipeline before moving to LVIS segmentation.

3.5.2 Training LVISSegNet on LVIS

Complete `LVISSegNet` code (from `models.py`):

```
class SkipBlockDN(nn.Module):
    def __init__(self, in_ch: int, out_ch: int) -> None:
```

```

        super().__init__()
        self.conv1 = nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1,
bias=False)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.conv2 = nn.Conv2d(out_ch, out_ch, kernel_size=3, padding=1,
bias=False)
        self.bn2 = nn.BatchNorm2d(out_ch)
        self.skip = nn.Conv2d(in_ch, out_ch, kernel_size=1, bias=False) if
in_ch != out_ch else nn.Identity()
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

    def forward(self, x: torch.Tensor) -> tuple[torch.Tensor,
torch.Tensor]:
        identity = self.skip(x)
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out = F.relu(out + identity)
        pooled = self.pool(out)
        return pooled, out

class SkipBlockUP(nn.Module):
    def __init__(self, in_ch: int, out_ch: int) -> None:
        super().__init__()
        self.up = nn.ConvTranspose2d(in_ch, out_ch, kernel_size=2,
stride=2)
        self.conv1 = nn.Conv2d(out_ch + in_ch, out_ch, kernel_size=3,
padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.conv2 = nn.Conv2d(out_ch, out_ch, kernel_size=3, padding=1,
bias=False)
        self.bn2 = nn.BatchNorm2d(out_ch)

    def forward(self, x: torch.Tensor, skip: torch.Tensor) -> torch.Tensor:
        x = self.up(x)
        x = torch.cat([x, skip], dim=1)
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        return x

class LVISSegNet(nn.Module):
    def __init__(self, num_classes: int = 4, dropout_p: float = 0.1) ->
None:
        super().__init__()
        self.enc1 = SkipBlockDN(3, 64)
        self.enc2 = SkipBlockDN(64, 128)
        self.enc3 = SkipBlockDN(128, 256)
        self.enc4 = SkipBlockDN(256, 512)

        self.bottleneck = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(512),

```

```

        nn.ReLU(inplace=True),
    )
    self.dropout = nn.Dropout2d(p=dropout_p)

    self.dec4 = SkipBlockUP(512, 256)
    self.dec3 = SkipBlockUP(256, 128)
    self.dec2 = SkipBlockUP(128, 64)
    self.dec1 = SkipBlockUP(64, 64)
    self.final = nn.Conv2d(64, num_classes, kernel_size=1)

def forward(self, x: torch.Tensor) -> torch.Tensor:
    x, s1 = self.enc1(x)
    x, s2 = self.enc2(x)
    x, s3 = self.enc3(x)
    x, s4 = self.enc4(x)
    x = self.bottleneck(x)
    x = self.dropout(x)
    x = self.dec4(x, s4)
    x = self.dec3(x, s3)
    x = self.dec2(x, s2)
    x = self.dec1(x, s1)
    return self.final(x)

```

Parameter count:

- **LVISegNet: 13,494,660** learnable parameters

Training and validation loss curves

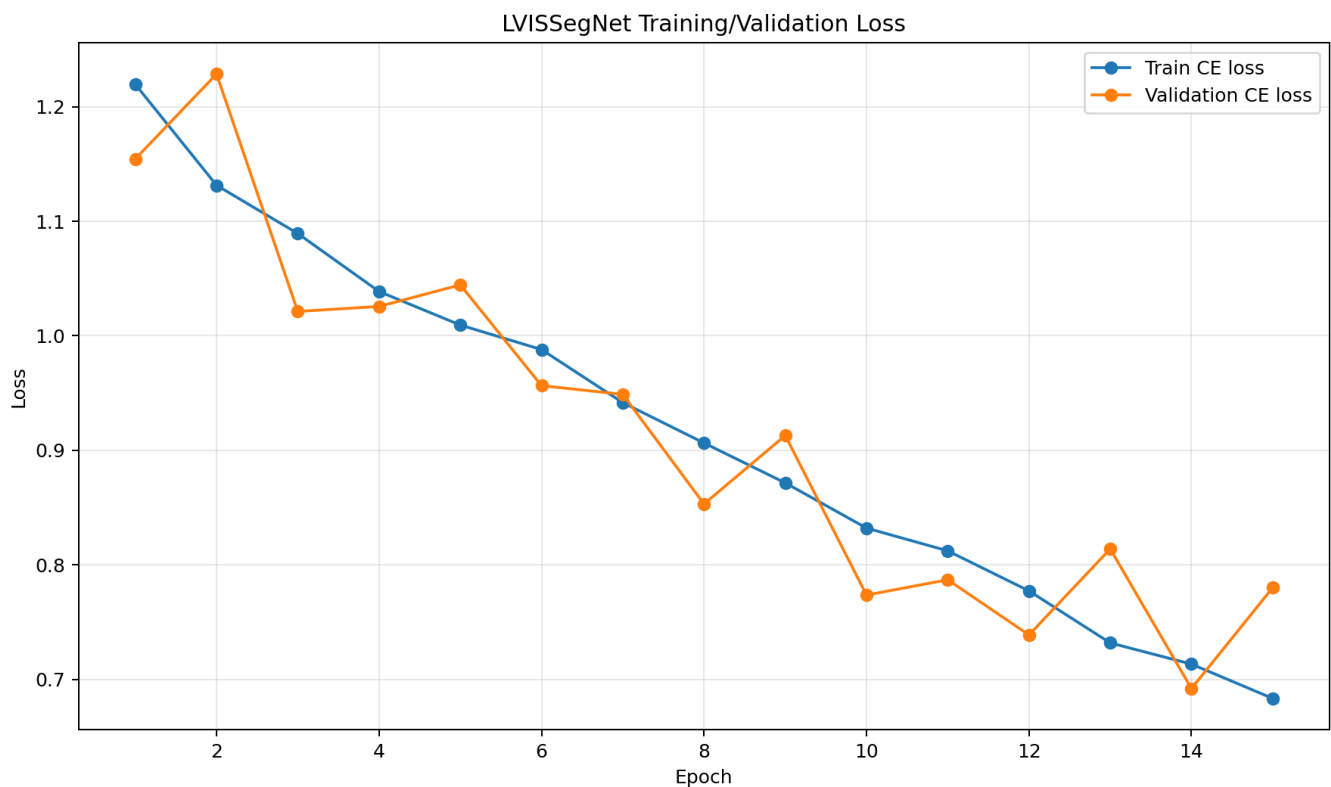
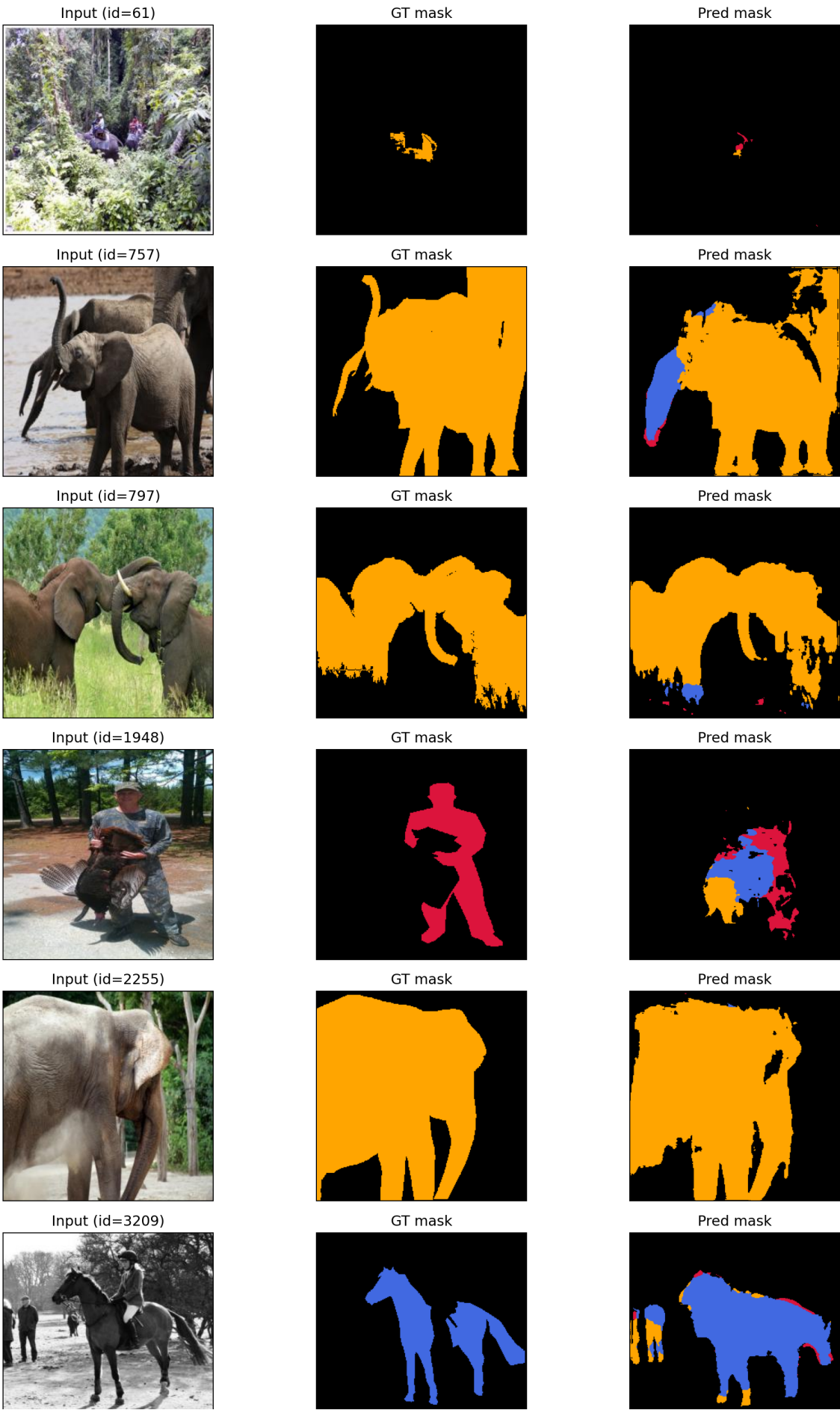


Figure 13. Task 5.2 LVISegNet training and validation loss curves (15-epoch GPU run).

Validation triplets (input / GT mask / predicted mask)

Task 3.5.2 Validation Triplets (Input / GT / Prediction)



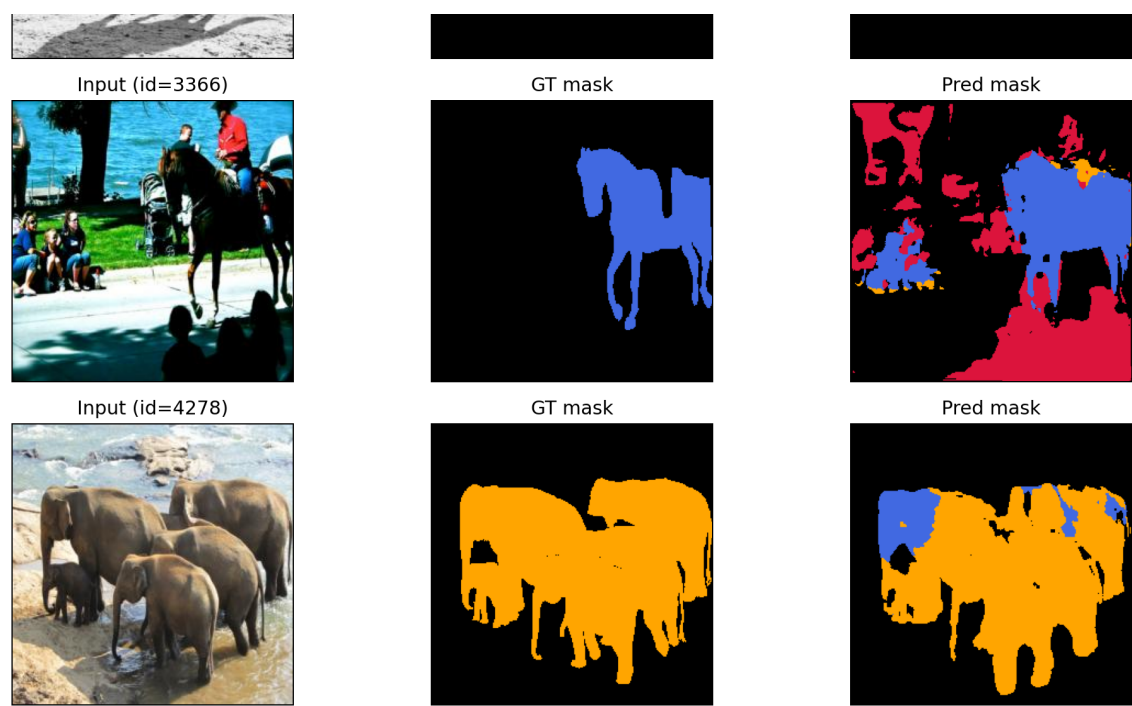


Figure 14. Task 5.2 validation triplets from the full validation set (981 images).

Table 1. Task 5.2 per-class pixel accuracy and IoU computed on the full validation set.

Class	Pixel Accuracy	IoU
background	0.8270	0.7931
person	0.6771	0.2656
horse	0.5778	0.2981
elephant	0.6880	0.5090

Overall and mIoU metrics:

- Overall pixel accuracy: 0.7933
- Foreground mIoU (person/horse/elephant): 0.3576

Performance paragraph: LVISSegNet segments background reliably and captures coarse foreground regions, with overall pixel accuracy 0.7933 on the full validation set. Foreground mIoU 0.3576 indicates that person/horse/elephant boundaries remain imperfect, especially in crowded scenes and near fine structures. Compared with background, foreground classes are harder due to class imbalance and small-object detail, which leads to blurrier edges and occasional class confusion.

3.5.3 Quantitative Evaluation and Master Summary

Evaluation code snippet:

```
pred_boxes, pred_labels, pred_scores = decode_yolo_output(  
    output=pred_flat,  
    img_size=args.img_size,  
    grid_size=args.grid_size,
```

```
        anchor_ratios=DEFAULT_ANCHOR_RATIOS,
        num_classes=args.num_classes,
        conf_thresh=args.conf_thresh,
    )
    pred_boxes, pred_labels, pred_scores = nms_per_class(
        pred_boxes, pred_labels, pred_scores, iou_thresh=args.nms_iou_thresh
    )
    metrics = match_predictions(
        pred_boxes=pred_boxes,
        pred_labels=pred_labels,
        pred_scores=pred_scores,
        gt_boxes=gt_boxes,
        gt_labels=gt_labels,
        iou_thresh=args.match_iou_thresh,
    )
```

Table 2. Task 5.3 per-class precision and recall at IoU ≥ 0.5 .

Class	Precision	Recall
person	0.0000	0.0000
horse	0.5000	0.0036
elephant	0.8654	0.0681

Table 3. Master summary table.

Network	Dataset	Parameters	Key Metric
LOADnet2	PurdueShapes5	21,548,409	Classification accuracy: 88.9%
NetForYOLO	PurdueDrEvalMulti	54,264,340	P=0.393, R=0.440 (IoU $\geq$ 0.5); Dr_Eval R=0.61, house R=0.60, watertower R=0.10
LVISYOLONet	LVIS (person/horse/elephant)	21,495,360	person P=0.00/R=0.00; horse P=0.50/R=0.004; elephant P=0.87/R=0.068
mUnet	PurdueShapes5MultiObject	6,999,109	Pixel acc (4 test imgs): 0.9001, fg mIoU: 0.238
LVISegNet	LVIS (person/horse/elephant + bg)	13,494,660	Foreground mIoU: 0.3576

Final takeaways:

Detection is harder than classification: the model must localize a variable number of objects, which in the LVIS run yields high precision but very low recall — detections are conservative and small/crowded objects are missed. The three YOLO losses are complementary: objectness controls activation, classification assigns

category, and regression refines geometry; here, sparse positives and assignment collisions primarily suppressed recall.

Segmentation provides dense pixel-level structure that boxes cannot — LVISegNet encodes region boundaries and scene layout, while detection gives discrete instance identity. The two are complementary: segmentation is better for boundary/region understanding, detection for instance localization.

4 Bonus Task: DIoU Loss for Bounding Box Regression

Training config (matching Task 4):

- `epochs=15`, full training (no batch cap)
- full validation on all `981` validation images

Bonus training command:

- `python -u run_hw6_bonus_diou_train_eval.py --device cuda --epochs 15 --batch-size 16 --lr 1e-4 --grid-size 8 --img-size 256 --conf-thresh 0.35 --nms-iou-thresh 0.5 --match-iou-thresh 0.5 --num-workers 0 --log-every 20 --val-log-every 100 2>&1 | tee results/hw6_bonus_diou/bonus_diou_train_log.txt`

DIoU implementation code:

```
def compute_diou_loss(pred_boxes, target_boxes, reduction="mean", eps=1e-6):
    pred_x1y1, pred_x2y2 = _xywh_to_xyxy_torch(pred_boxes)
    target_x1y1, target_x2y2 = _xywh_to_xyxy_torch(target_boxes)

    inter_x1y1 = torch.maximum(pred_x1y1, target_x1y1)
    inter_x2y2 = torch.minimum(pred_x2y2, target_x2y2)
    inter_wh = (inter_x2y2 - inter_x1y1).clamp(min=0)
    inter_area = inter_wh[..., 0] * inter_wh[..., 1]

    pred_wh = (pred_x2y2 - pred_x1y1).clamp(min=0)
    target_wh = (target_x2y2 - target_x1y1).clamp(min=0)
    union_area = (pred_wh[..., 0] * pred_wh[..., 1] + target_wh[..., 0] *
target_wh[..., 1] - inter_area).clamp(min=eps)
    iou = inter_area / union_area

    pred_centers = pred_x1y1 + pred_wh / 2.0
    target_centers = target_x1y1 + target_wh / 2.0
    center_dist_sq = ((pred_centers - target_centers) ** 2).sum(dim=-1)

    enclose_x1y1 = torch.minimum(pred_x1y1, target_x1y1)
    enclose_x2y2 = torch.maximum(pred_x2y2, target_x2y2)
    diag_sq = (((enclose_x2y2 - enclose_x1y1).clamp(min=0)) **
2).sum(dim=-1).clamp(min=eps)
    loss = 1.0 - iou + center_dist_sq / diag_sq
    return loss.mean() if reduction == "mean" else loss
```

Training-loop loss swap:


```

pred_boxes = offsets_to_boxes(
    pred[..., 1 + num_classes :],
    cell_centers=cell_centers,
    anchor_wh=anchor_wh,
    cell_size=cell_size,
    scale_mode="softplus",
)
target_boxes = offsets_to_boxes(
    gt[..., 1 + num_classes :],
    cell_centers=cell_centers,
    anchor_wh=anchor_wh,
    cell_size=cell_size,
    scale_mode="clamp",
)
bbox_loss = compute_diou_loss(pred_boxes[pos_mask], target_boxes[pos_mask],
    reduction="mean")
total_loss = obj_loss + cls_loss + bbox_loss

```

Training loss curves for the DIoU variant

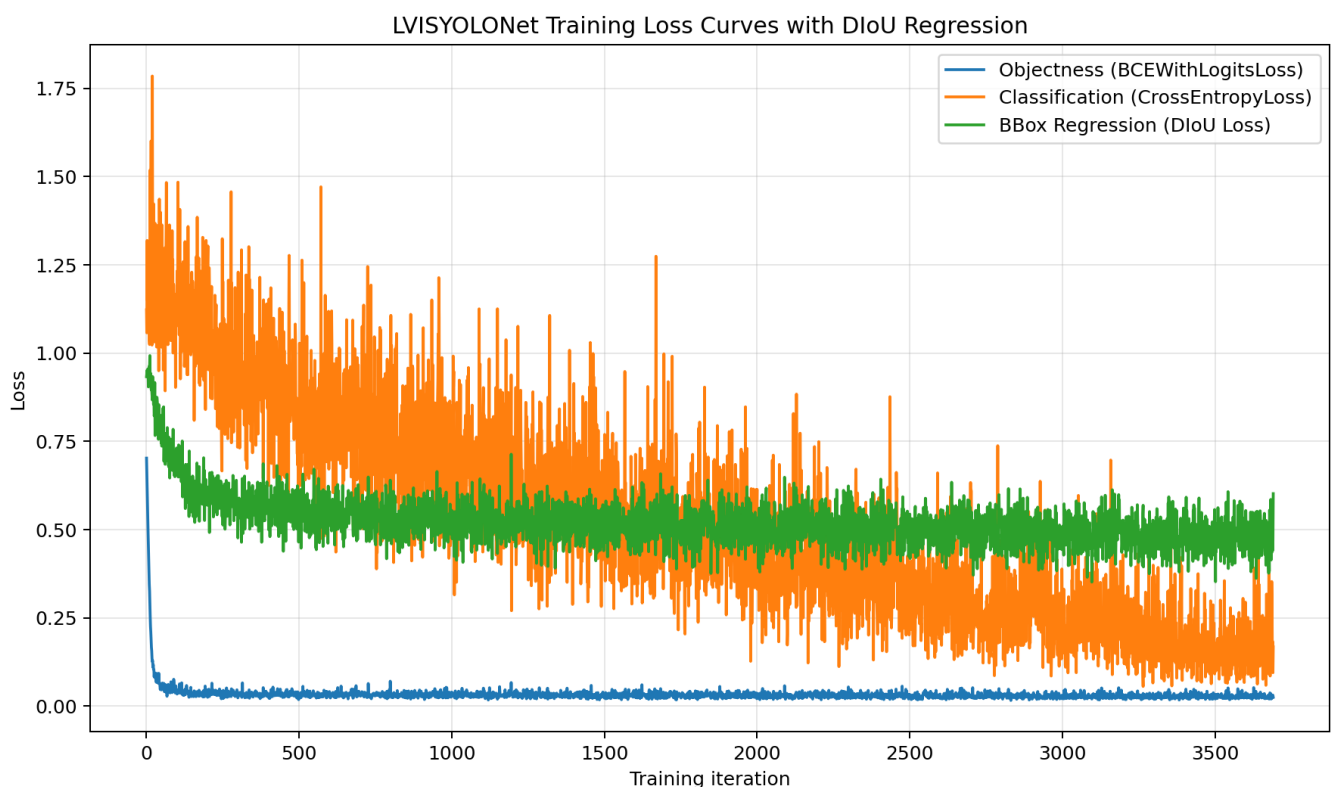


Figure 15. Bonus DIoU run training loss curves for objectness, classification, and DIoU regression.

From the training log, the DIoU regression term decreased from about 0.94 at epoch 1 to about 0.48 by epoch 15, confirming the DIoU objective was being optimized throughout.

12 annotated validation images

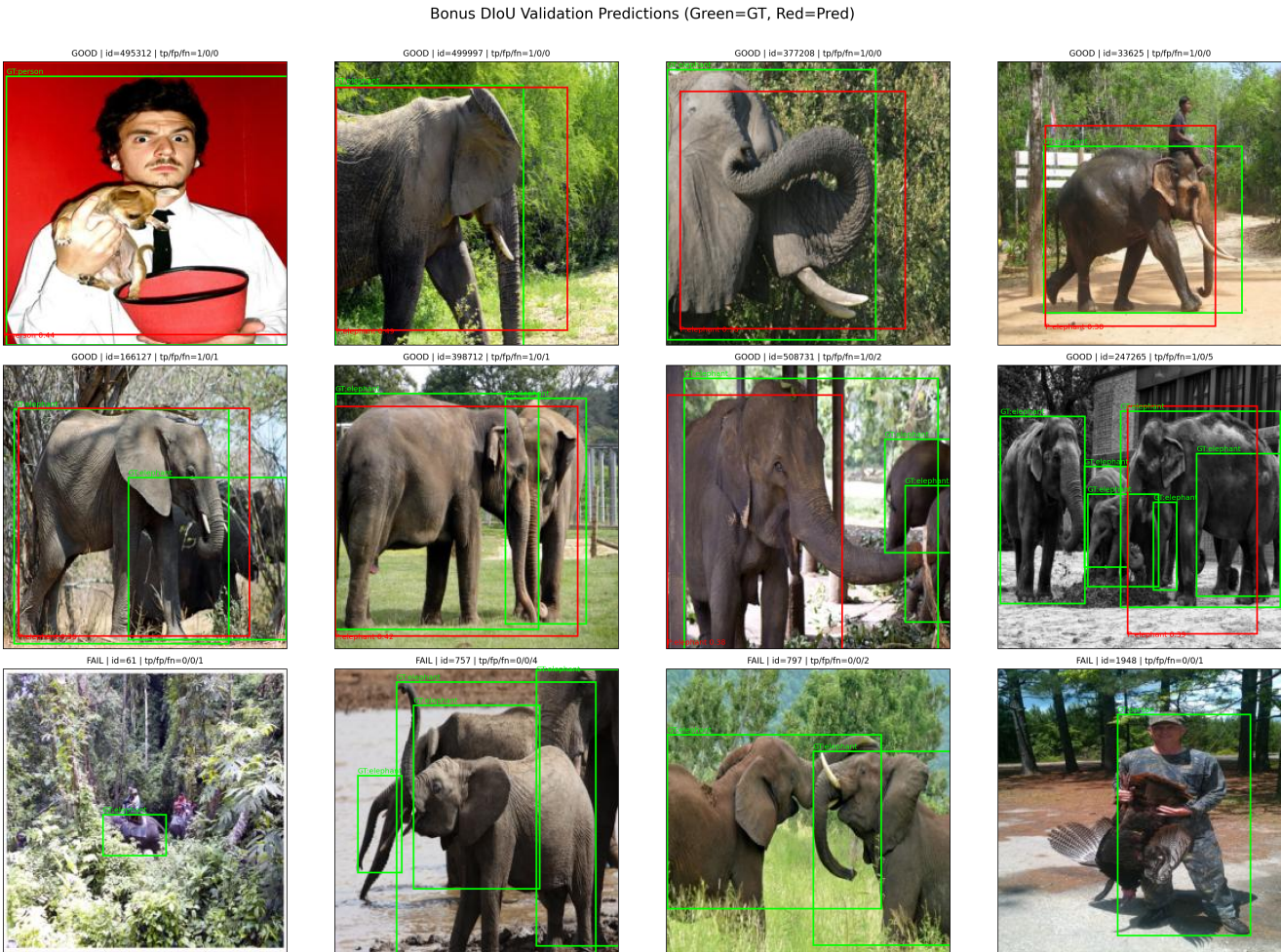


Figure 16. Bonus DIoU validation predictions (Green=GT, Red=Pred) with 8 good and 4 failure cases.

Table 4. MSE-based regression vs. DIoU-based regression at $\text{IoU} \geq 0.5$.

Class	MSE Precision	MSE Recall	DIoU Precision	DIoU Recall	Delta Precision	Delta Recall
person	0.0000	0.0000	1.0000	0.0015	+1.0000	+0.0015
horse	0.5000	0.0036	0.0000	0.0000	-0.5000	-0.0036
elephant	0.8654	0.0681	1.0000	0.0106	+0.1346	-0.0575
overall	0.8246	0.0249	0.8889	0.0042	+0.0643	-0.0206

Bonus discussion:

Replacing MSE with DIoU raised overall precision from 0.825 to 0.889 but reduced recall from 0.025 to 0.004 at $\text{IoU} \geq 0.5$. The DIoU model produces fewer but more accurate detections: person and elephant precision reached 1.0, while horse detections dropped to zero. This pattern suggests DIoU produces tighter boxes that only clear the IoU threshold when the center penalty is well-minimized, making the model more conservative. Recall remains the bottleneck, driven by weak objectness confidence and grid/anchor assignment limits rather than the regression loss itself.