

ECE60146 / BME646 HW6 Report

Student: Weiyi Chen

Chosen LVIS classes:

- `car_(automobile)`
- `chair`
- `person`

These three classes meet the homework constraints (enough qualifying instances, multi-instance scenes, and area threshold) and provide a reasonable mix of rigid and non-rigid objects.

1) Task 1: Single-Instance Detection Warm-up

1(a) Code used to run LOADnet2

Code snippet from `task1_single_instance.py`:

```
dls = DLStudio(
    dataroot="data/PurdueShapes5/",
    image_size=[32, 32],
    path_saved_model=os.path.join(OUT_DIR, "loadnet2.pt"),
    momentum=0.9,
    learning_rate=1e-4,
    epochs=epochs,
    batch_size=4,
    classes=("rectangle", "triangle", "disk", "oval", "star"),
    use_gpu=torch.cuda.is_available(),
)
detector = DLStudio.DetectAndLocalize(dl_studio=dls)
model = detector.LOADnet2(skip_connections=True, depth=8)
detector.run_code_for_training_with_CrossEntropy_and_MSE_Losses(model,
    show_images=False)
```

This experiment follows the DLStudio warm-up pipeline [1].

I kept the default warm-up setting (32x32, batch size 4, low learning rate) to verify the baseline pipeline before moving to LVIS-scale tasks.

1(b) Figure: training loss curves

Figure 1. Task 1 dual-loss training curves.

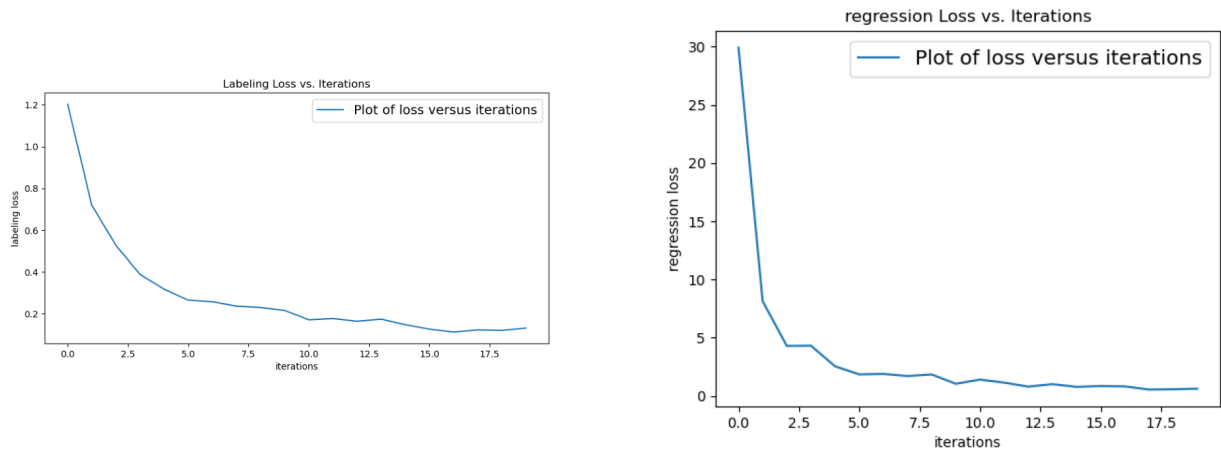


Figure 2. Task 1 classification loss.

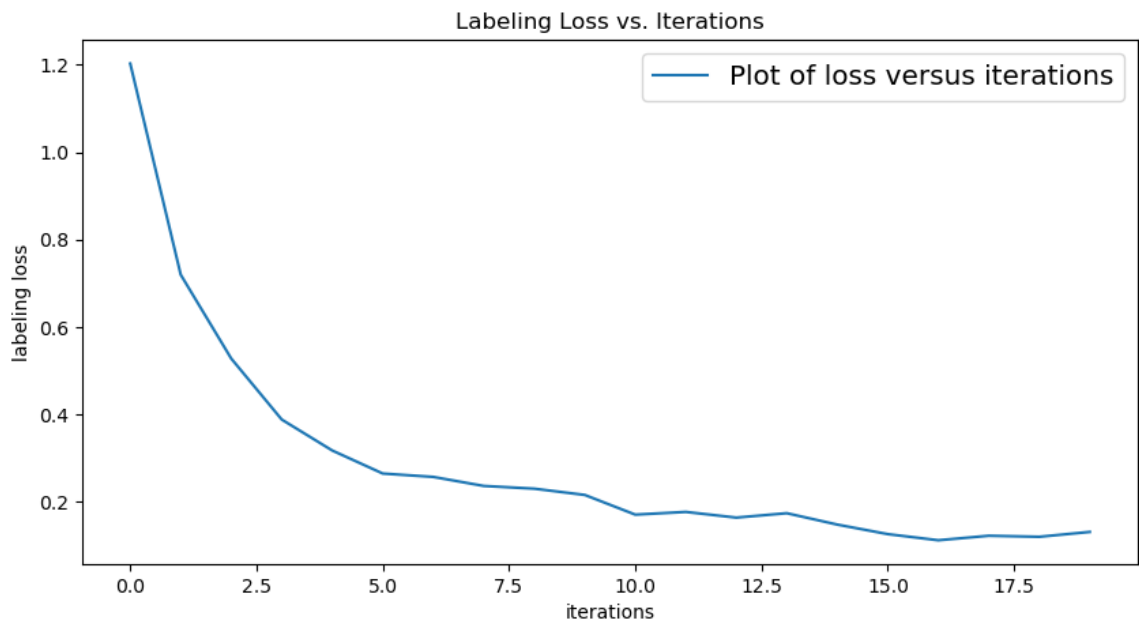
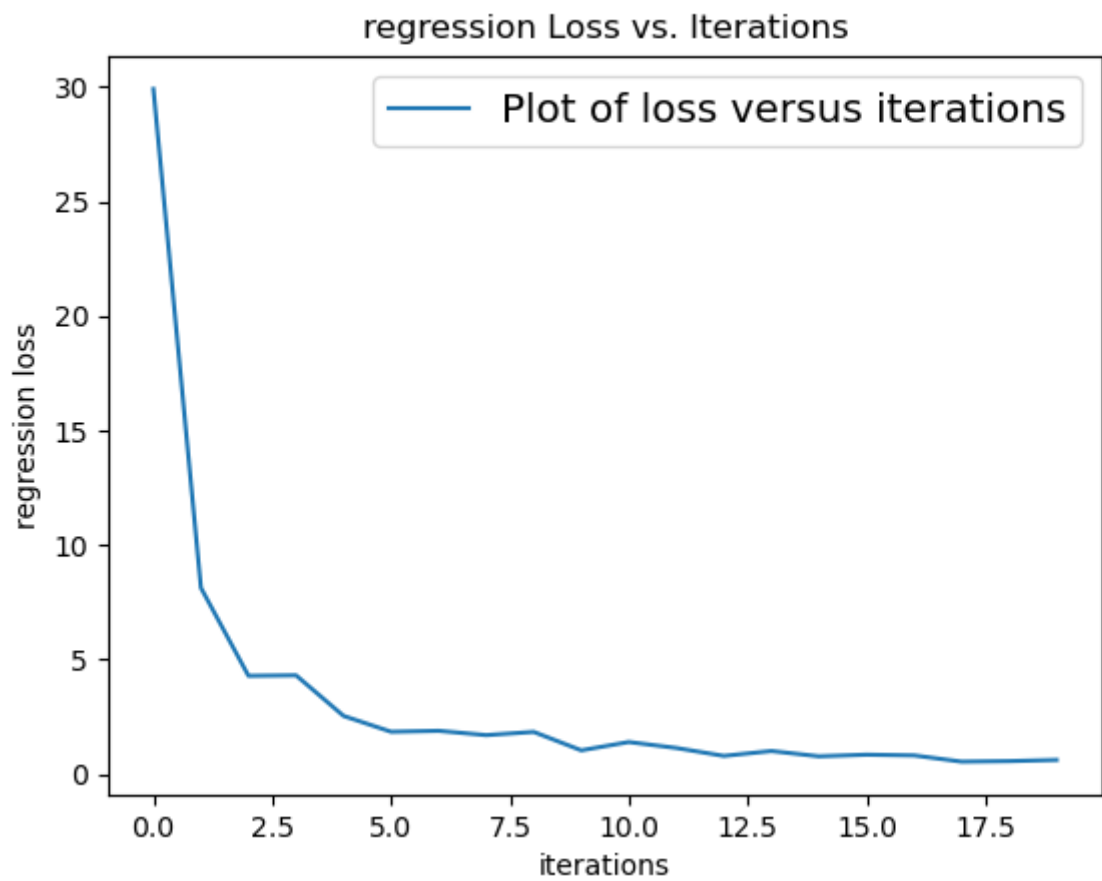


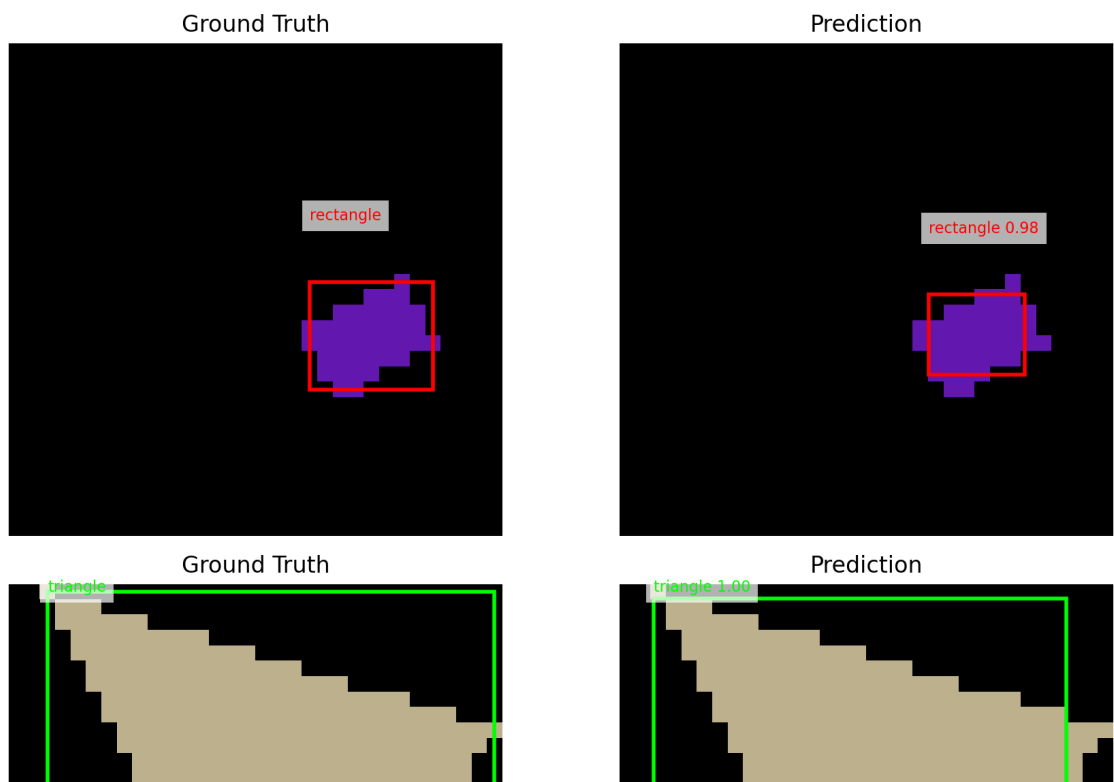
Figure 3. Task 1 box regression loss.

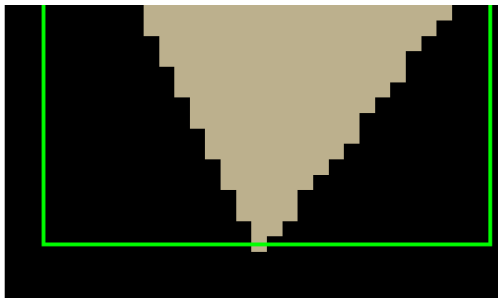


The curves show fast improvement in classification and a slower but stable decrease in localization error, which is expected for joint classification-regression training.

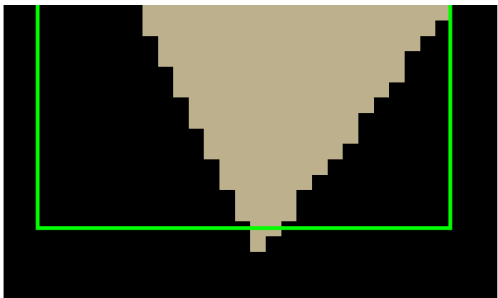
1(c) Figure: test grid with GT and predictions

Figure 4. Task 1 test visualization (≥ 6 images).

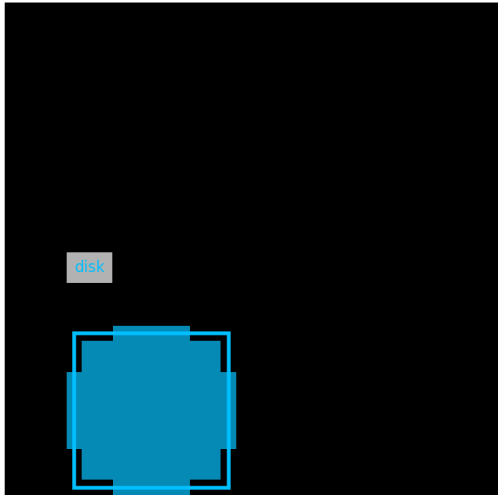




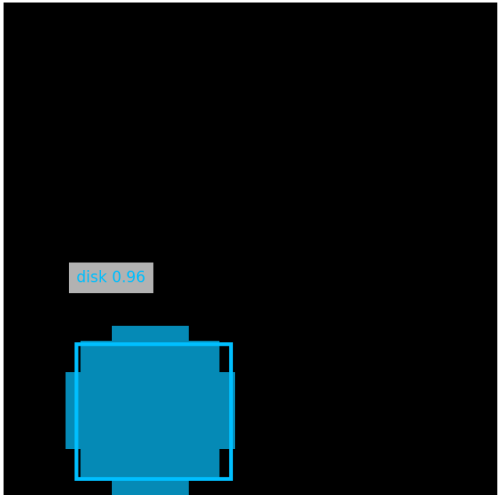
Ground Truth



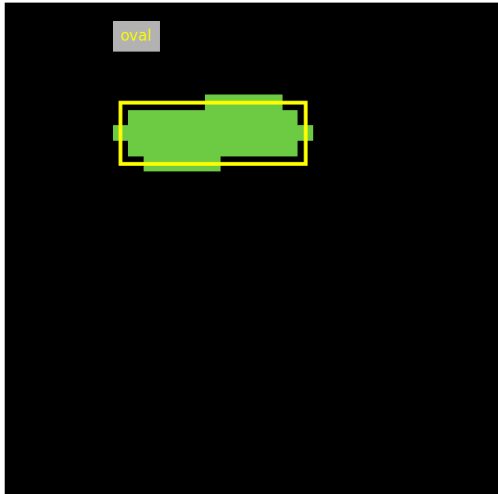
Prediction



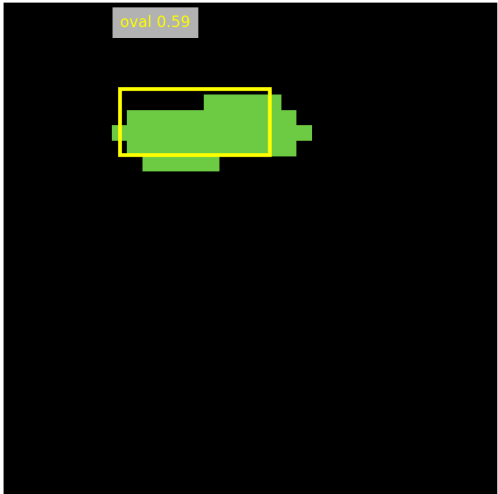
Ground Truth



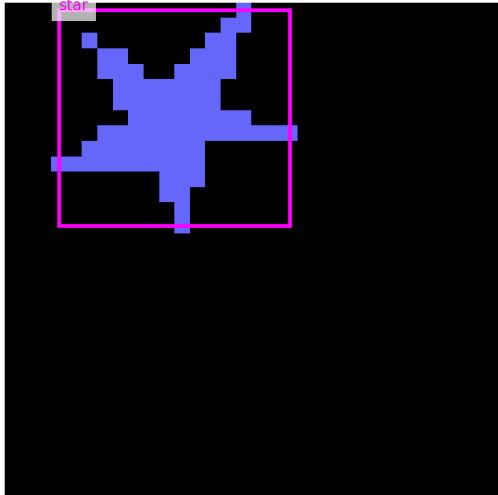
Prediction



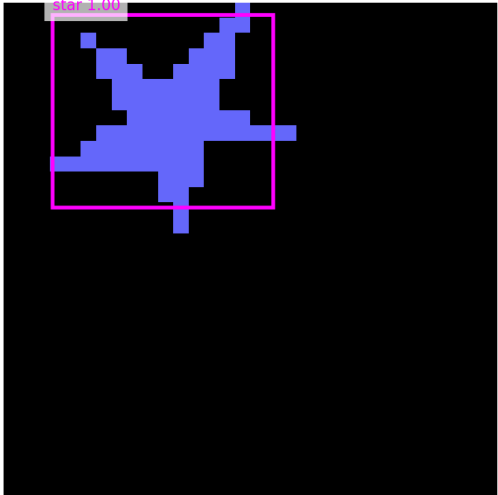
Ground Truth



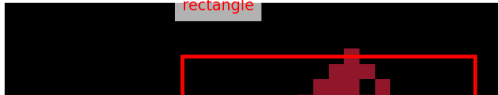
Prediction



Ground Truth



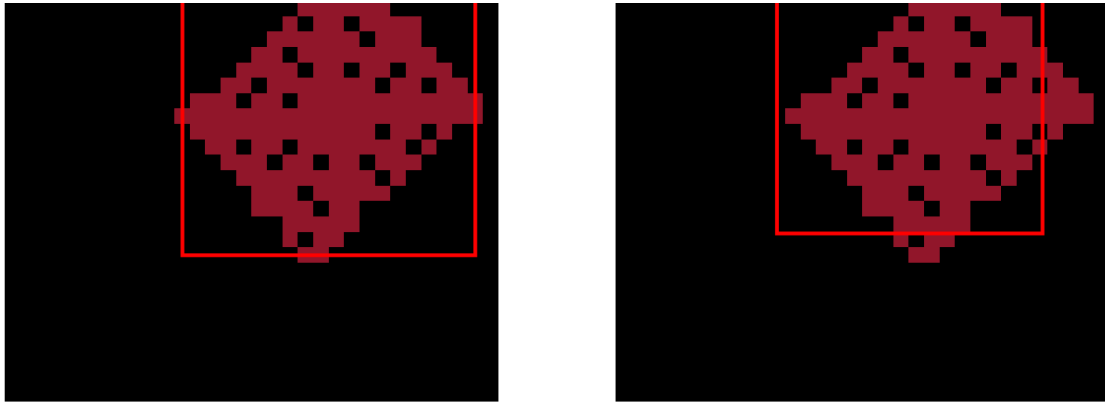
Prediction



Ground Truth



Prediction



Most predicted boxes align tightly with the object boundaries in this simple dataset, matching the high classification accuracy.

1(d) Classification accuracy

Test accuracy is **100.00%**.

1(e) Dual-loss explanation (branching, retain_graph, MSE vs CE)

`LOADnet2` uses a shared backbone and then splits into two heads: one for class logits and one for box regression. This is why the model can optimize recognition and localization jointly.

If the two losses are backpropagated separately, the first backward pass frees the graph. In that case `retain_graph=True` is required for the first backward call. If the two losses are summed and backpropagated once, `retain_graph` is not needed.

`CrossEntropyLoss` is used for class prediction because labels are discrete categories. `MSELoss` is used for box coordinates because localization targets are continuous values.

2) Task 2: Understanding YOLO Logic

2(a) Text: cells, anchors, yolo_vector, yolo_tensor

YOLO divides each image into grid cells. A cell is responsible for objects whose centers fall inside that cell [2,5]. This converts variable-length annotations into a fixed-size prediction tensor.

Each cell predicts multiple anchors to cover different shapes. During target assignment, the anchor with the best match to the GT box (typically highest IoU) is selected [2,5].

`yolo_vector` structure in this homework:

- objectness (1)
- class logits/probabilities ($K=3$)
- box offsets (`dx`, `dy`, `sw`, `sh`) (4)

So vector length is $V = 1 + K + 4 = 8$.

Equation (1): center offsets used in assignment/decoding.

$dx = (cx_{gt} - cx_{cell}) / cell_size, dy = (cy_{gt} - cy_{cell}) / cell_size$

Equation (2): size relative to anchor.

$w_{gt} = sw * w_{anchor}, h_{gt} = sh * h_{anchor}$

yolo_tensor layout: (B, C, A, V) where B batch, C cells, A anchors/cell, V vector length.

2(b) Code: script used for multi-instance warm-up

```

yolomod = YOLOLogic(
    dataroot_train=cfg.dataroot_train,
    dataroot_test=cfg.dataroot_test,
    image_size=list(cfg.image_size),
    yolo_interval=cfg.yolo_interval,
    learning_rate=cfg.lr,
    epochs=cfg.epochs,
    batch_size=cfg.batch_size,
    classes=("Dr_Eval", "house", "watertower"),
    use_gpu=torch.cuda.is_available(),
)
yolo = yolomod.YoloObjectDetector(yolomod=yolomod)
model = yolo.NetForYolo(skip_connections=True, depth=8).to(device)

```

This script configures the YOLOLogic toy dataset and then optimizes objectness, class, and box regression losses jointly.

Training loop core:

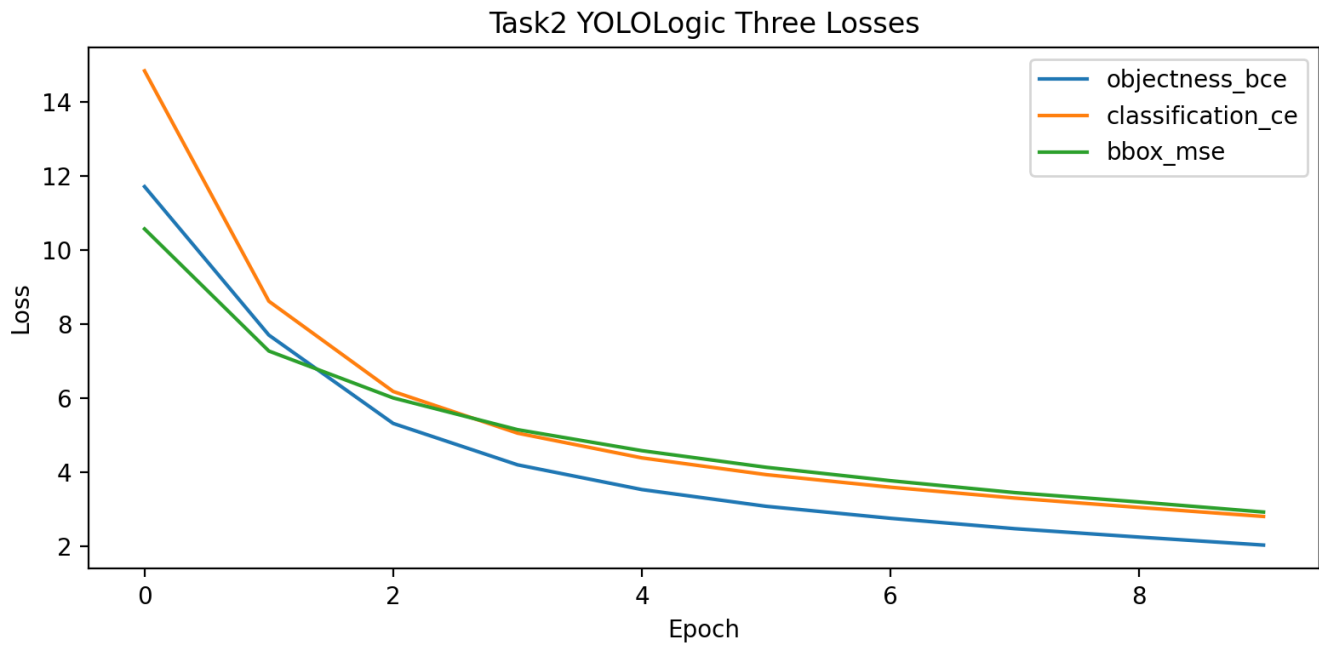
```

pred = out.view(-1, num_yolo_cells, num_anchor_boxes, 9)
obj = bce_loss(torch.sigmoid(pred[:, :, :, 0]), yolo_aug[:, :, :, 0])
reg = mse_loss(pred[:, :, :, 1:5], yolo_aug[:, :, :, 1:5])
targets = torch.argmax(yolo_aug[:, :, :, 5:].contiguous().view(-1, 4), dim=1)
cls = ce_loss(pred[:, :, :, 5:].contiguous().view(-1, 4), targets)
total = obj + reg + cls

```

2(c) Figure: three loss curves

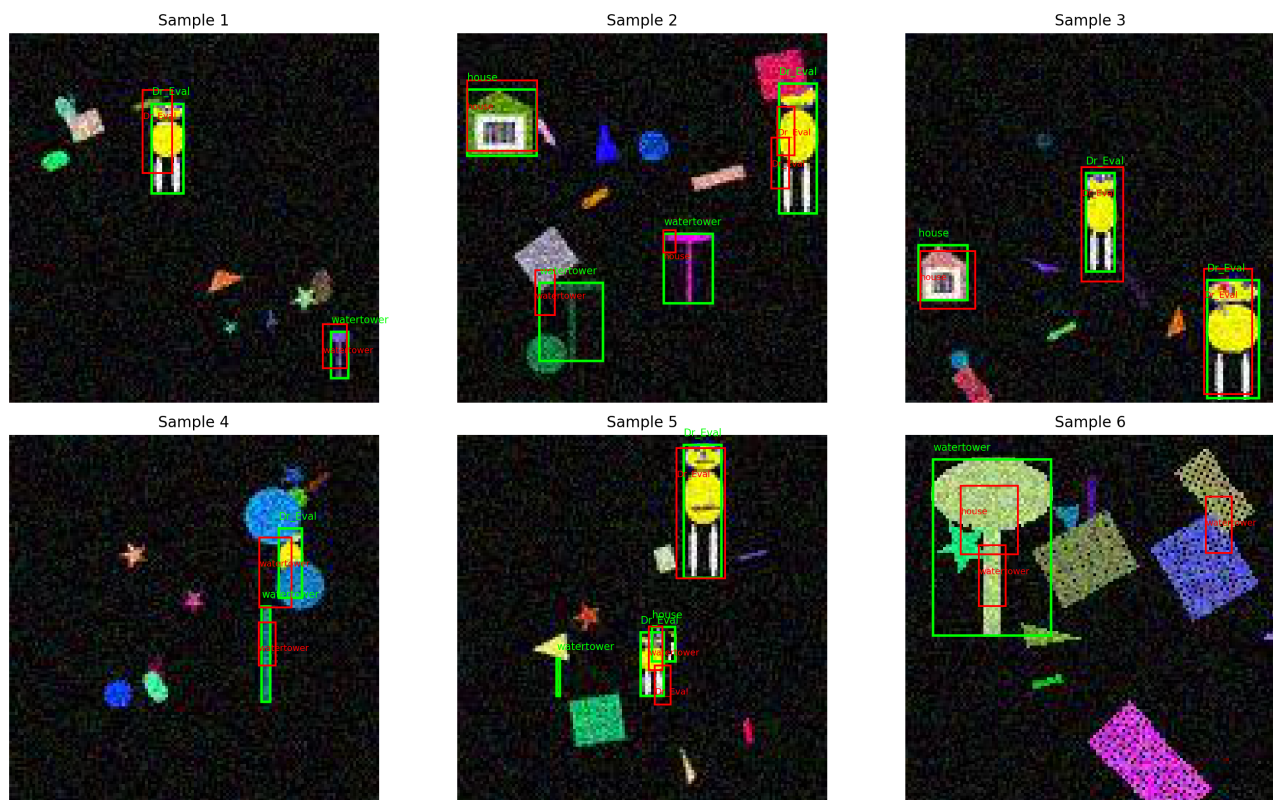
Figure 5. Task 2 YOLO warm-up losses (objectness/cls/regression).



All three losses decrease over training, with regression showing higher variance than objectness/classification due to harder localization targets.

2(d) Figure: test grid

Figure 6. Task 2 multi-instance predictions (≥ 6 images).



Qualitatively, the model detects multiple objects in the same image, but localization quality varies across categories.

2(e) Detection quality discussion

From `artifacts/task2/task2_metrics.json`, overall accuracy is **90.0%**. Per-class accuracy is `Dr_Eval` 88%, `house` 93%, and `watertower` 88%. The IoU trend (`0.597`, `0.508`, `0.238`) suggests `watertower` localization is the hardest category in noisy scenes.

3) Task 3: LVIS Dataset Construction (Detection + Segmentation) and IoU

3(a) Code: complete detection dataset construction script

Script entrypoint:

```
def run_task3():
    os.makedirs(OUT_DIR, exist_ok=True)
    cfg = BuildConfig(
        ann_file="annotations/lvis_v1_train.json",
        coco_train_dir="train2017",
        out_dir=DATA_ROOT,
        class_list=DEFAULT_CLASS_LIST,
    )
    stats = build_lvis_detection_and_segmentation_dataset(cfg)
    with open(os.path.join(OUT_DIR, "task3_dataset_stats.json"), "w",
              encoding="utf-8") as f:
        json.dump(stats, f, indent=2)

    _make_detection_grid(DEFAULT_CLASS_LIST)
    _make_segmentation_samples(DEFAULT_CLASS_LIST)
    _run_iou_tests()
```

Core detection+split construction (`dataset.py`):

```
def build_lvis_detection_and_segmentation_dataset(cfg: BuildConfig) -> Dict:
    random.seed(cfg.seed)
    np.random.seed(cfg.seed)
    os.makedirs(cfg.out_dir, exist_ok=True)
    coco_root = os.path.dirname(cfg.coco_train_dir)

    data = _load_lvis_json(cfg.ann_file)
    cat_id_to_idx, img_to_anns, img_id_to_info, valid_img_ids =
    _prepare_image_ann_index(
        data, cfg.class_list, cfg.min_area
    )

    random.shuffle(valid_img_ids)
    total_target = min(len(valid_img_ids), cfg.train_target_range[1] +
    cfg.val_target_range[1])
    picked = valid_img_ids[:total_target]

    split = int(len(picked) * cfg.train_ratio)
    train_ids = picked[:split]
    val_ids = picked[split:]
```

```

if len(train_ids) > cfg.train_target_range[1]:
    train_ids = train_ids[: cfg.train_target_range[1]]
if len(val_ids) > cfg.val_target_range[1]:
    val_ids = val_ids[: cfg.val_target_range[1]]

for split_name in ["train", "val"]:
    for subdir in ["images", "det_ann", "seg_masks"]:
        path = os.path.join(cfg.out_dir, split_name, subdir)
        os.makedirs(path, exist_ok=True)
        for fname in os.listdir(path):
            fpath = os.path.join(path, fname)
            if os.path.isfile(fpath):
                os.remove(fpath)

num_classes = len(cfg.class_list)
train_obj_count_per_class = [0 for _ in range(num_classes)]
val_obj_count_per_class = [0 for _ in range(num_classes)]
train_pixel_counts = np.zeros(num_classes + 1, dtype=np.int64)
val_pixel_counts = np.zeros(num_classes + 1, dtype=np.int64)
train_total_obj = 0
val_total_obj = 0

def process_ids(ids: List[int], split_name: str):
    nonlocal train_total_obj, val_total_obj
    processed_ids = []
    for img_id in ids:
        img_info = img_id_to_info[img_id]
        coco_url = img_info.get("coco_url", "")
        parts = coco_url.split("/")
        fname = parts[-1]
        split_folder = parts[-2] if len(parts) >= 2 else "train2017"
        img_path = os.path.join(coco_root, split_folder, fname)
        if not os.path.exists(img_path):
            continue
        anns = img_to_anns[img_id]

        resized_img, scaled_anns = resize_image_and_boxes(img_path, anns,
target_size=cfg.target_size)
        img_out = os.path.join(cfg.out_dir, split_name, "images", fname)
        det_out = os.path.join(cfg.out_dir, split_name, "det_ann",
fname.replace(".jpg", ".json"))
        mask_out = os.path.join(cfg.out_dir, split_name, "seg_masks",
fname.replace(".jpg", ".png"))

        resized_img.save(img_out)
        _save_detection_annotation(det_out, scaled_anns, cat_id_to_idx)
        processed_ids.append(int(img_id))

        mask = create_segmentation_mask(
            anns=scaled_anns,
            img_w=cfg.target_size,
            img_h=cfg.target_size,
            target_size=cfg.target_size,
            lvis_id_to_idx=cat_id_to_idx,

```

```

    )
    Image.fromarray(mask).save(mask_out)

    for ann in scaled_anns:
        cls_idx = cat_id_to_idx[ann["category_id"]]
        if split_name == "train":
            train_obj_count_per_class[cls_idx] += 1
            train_total_obj += 1
        else:
            val_obj_count_per_class[cls_idx] += 1
            val_total_obj += 1
    counts = np.bincount(mask.reshape(-1), minlength=num_classes + 1)
    if split_name == "train":
        np.add(train_pixel_counts, counts, out=train_pixel_counts)
    else:
        np.add(val_pixel_counts, counts, out=val_pixel_counts)
    return processed_ids

processed_train_ids = process_ids(train_ids, "train")
processed_val_ids = process_ids(val_ids, "val")

train_images = len([f for f in os.listdir(os.path.join(cfg.out_dir, "train",
"images")) if f.endswith(".jpg")])
val_images = len([f for f in os.listdir(os.path.join(cfg.out_dir, "val",
"images")) if f.endswith(".jpg")])

train_pixel_total = max(1, int(train_pixel_counts.sum()))
val_pixel_total = max(1, int(val_pixel_counts.sum()))

train_pixel_distribution = {"background": float(train_pixel_counts[0] /
train_pixel_total)}
val_pixel_distribution = {"background": float(val_pixel_counts[0] /
val_pixel_total)}
for cls_idx, cls_name in enumerate(cfg.class_list, start=1):
    train_pixel_distribution[cls_name] = float(train_pixel_counts[cls_idx] /
train_pixel_total)
    val_pixel_distribution[cls_name] = float(val_pixel_counts[cls_idx] /
val_pixel_total)

image_stats = {
    "class_list": list(cfg.class_list),
    "train_images": train_images,
    "val_images": val_images,
    "train_instances_per_class": {cfg.class_list[i]:
int(train_obj_count_per_class[i]) for i in range(len(cfg.class_list))},
    "val_instances_per_class": {cfg.class_list[i]:
int(val_obj_count_per_class[i]) for i in range(len(cfg.class_list))},
    "train_avg_objects_per_image": float(train_total_obj / max(1,
train_images)),
    "val_avg_objects_per_image": float(val_total_obj / max(1, val_images)),
    "pixel_distribution": train_pixel_distribution,
    "pixel_distribution_train": train_pixel_distribution,
    "pixel_distribution_val": val_pixel_distribution,
    "meets_hw6_size_target": {

```



```
        "train": bool(cfg.train_target_range[0] <= train_images <=
cfg.train_target_range[1]),
        "val": bool(cfg.val_target_range[0] <= val_images <=
cfg.val_target_range[1]),
    },
}

with open(os.path.join(cfg.out_dir, "dataset_stats.json"), "w", encoding="utf-
8") as f:
    json.dump(image_stats, f, indent=2)

with open(os.path.join(cfg.out_dir, "split_manifest.json"), "w",
encoding="utf-8") as f:
    json.dump({"train_image_ids": processed_train_ids, "val_image_ids":
processed_val_ids}, f, indent=2)

return image_stats
```

This construction pipeline does four things in one pass: (1) filters LVIS annotations by class and area, (2) creates a train/val split, (3) writes detection labels and segmentation masks, and (4) records dataset statistics for reproducibility.

3(b) Output: dataset verification

Table 1. LVIS dataset verification output

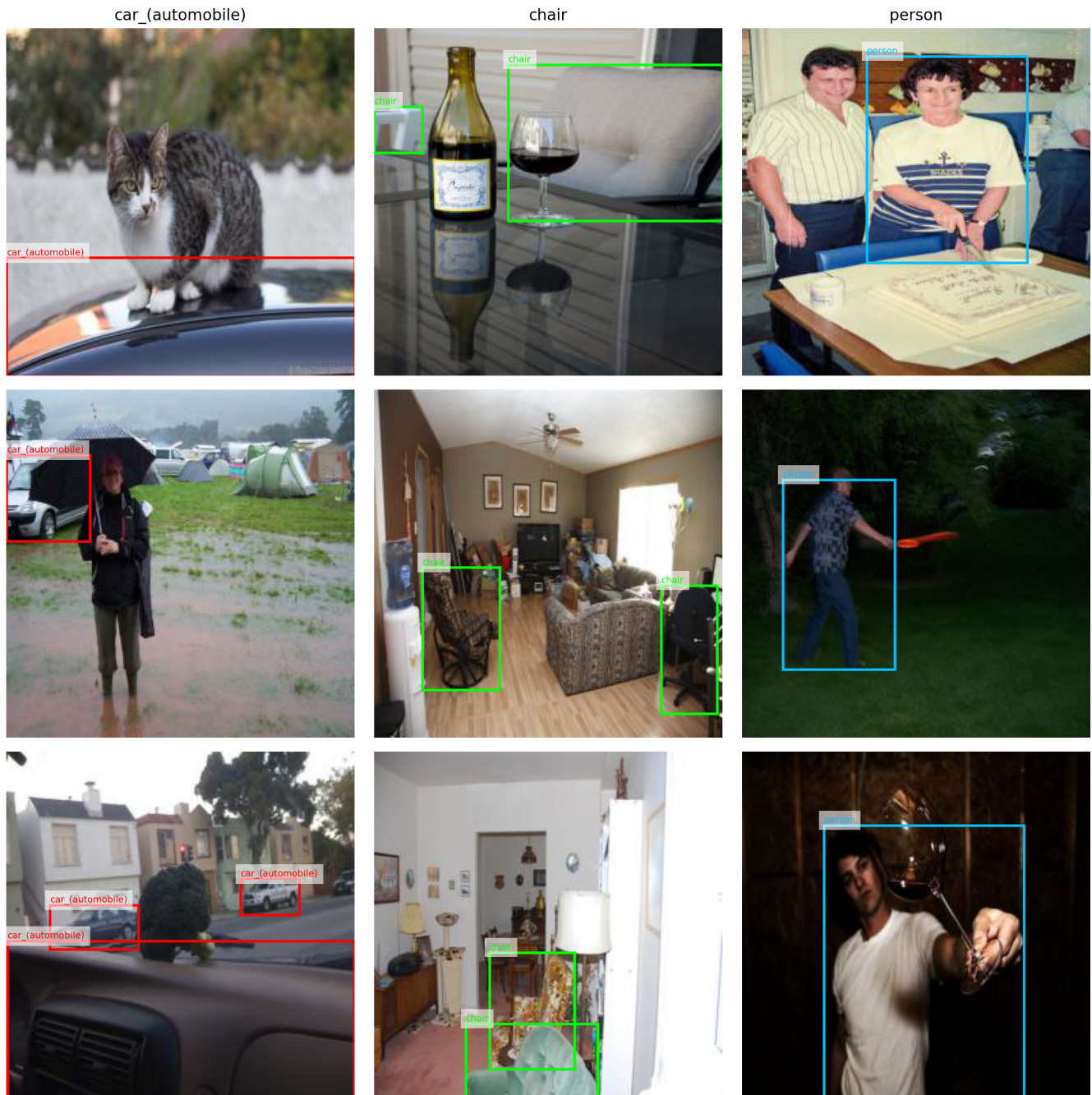
Item	Value
Train images	3264
Validation images	817
Train instances (car)	1528
Train instances (chair)	2145
Train instances (person)	2662
Train avg objects/image	1.9409

Selected class names: `car_(automobile)`, `chair`, `person`.

The final split (`3264/817`) is inside the required range, and the average object count confirms the dataset is truly multi-instance.

3(c) Figure: 3x3 detection sample grid

Figure 7. Task 3 detection verification grid (3x3).



This grid checks that resized images and transformed boxes remain aligned after preprocessing.

3(d) Code: segmentation mask construction

Code from `dataset.py`:

```
def create_segmentation_mask(anns, img_w, img_h, target_size, lvis_id_to_idx):
    mask = Image.new("L", (img_w, img_h), 0)
    draw = ImageDraw.Draw(mask)

    for ann in anns:
        cat_id = ann["category_id"]
        if cat_id not in lvis_id_to_idx:
            continue
        cls_idx = lvis_id_to_idx[cat_id] + 1
        seg = ann.get("segmentation", [])
```

```

if not isinstance(seg, list):
    continue
for polygon in seg:
    if not isinstance(polygon, list) or len(polygon) < 6:
        continue
    poly_xy = list(zip(polygon[0::2], polygon[1::2]))
    draw.polygon(poly_xy, fill=int(cls_idx))

mask = mask.resize((target_size, target_size), Image.NEAREST)
return np.array(mask, dtype=np.uint8)

```

`Image.NEAREST` is important here because mask labels are discrete class IDs; bilinear interpolation would create invalid mixed label values.

3(e) Figure: at least 6 image/mask examples

Figure 8. Task 3 segmentation samples (≥ 6 , two per class target).

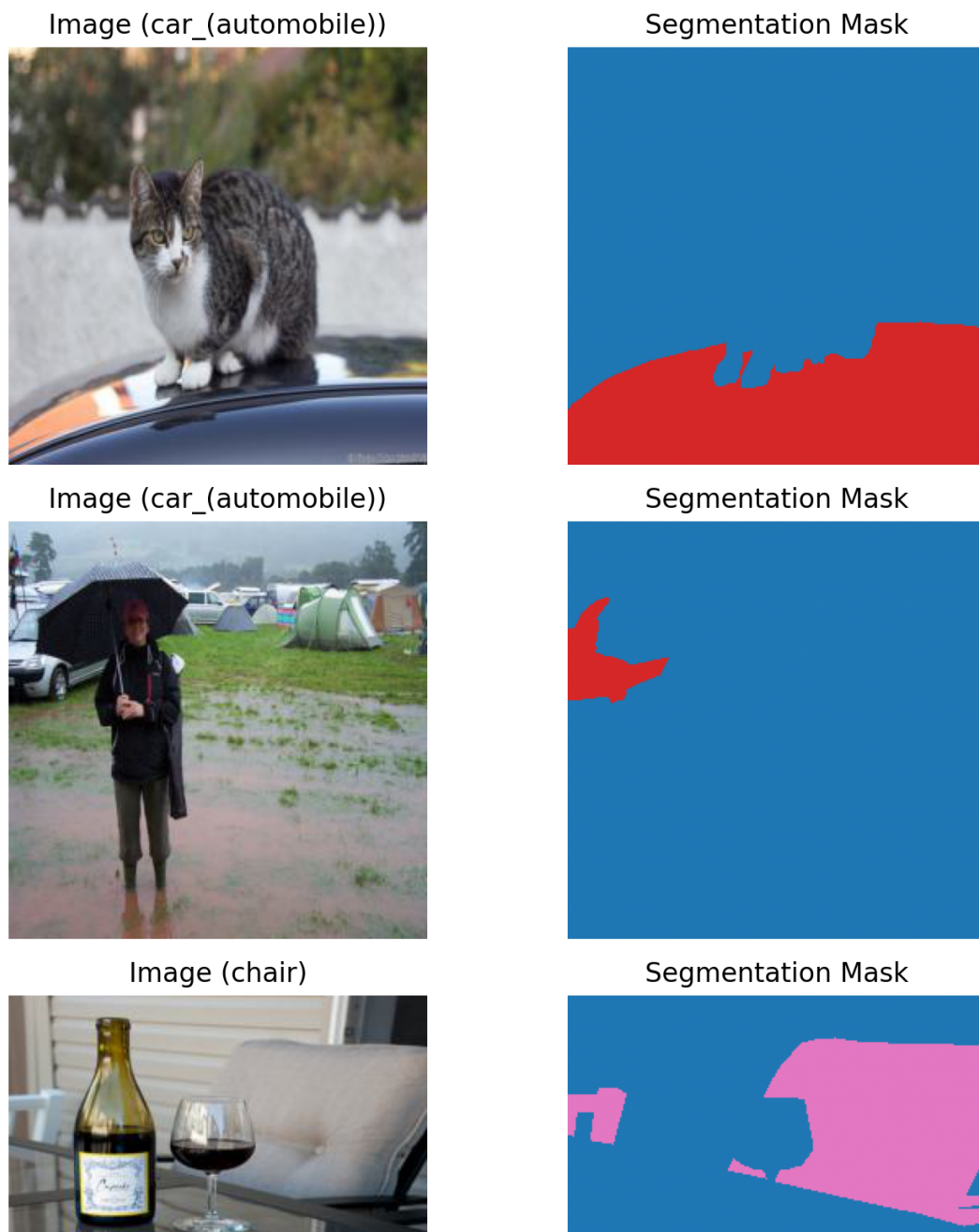




Image (chair)



Segmentation Mask

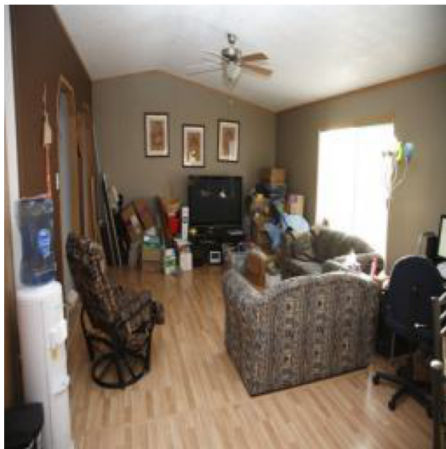


Image (person)



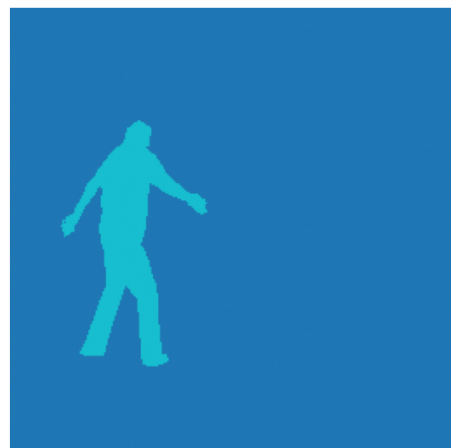
Segmentation Mask



Image (person)



Segmentation Mask



The samples show clear class regions for large objects, while thin boundaries and small objects are more difficult.

3(f) Text: per-class pixel distribution

From Task 3 stats:

- background: 0.8666
- car: 0.0337
- chair: 0.0271
- person: 0.0726

This strong imbalance explains why both detection and segmentation are more stable on dominant/background regions than on minority foreground classes.

3(g) Code: IoU implementation (`compute_iou`, `compute_iou_matrix`)

Code from `evaluate.py`:

```
def compute_iou(box1: Sequence[float], box2: Sequence[float]) -> float:
    x1, y1, w1, h1 = [float(v) for v in box1]
    x2, y2, w2, h2 = [float(v) for v in box2]

    xa1, ya1, xa2, ya2 = x1, y1, x1 + w1, y1 + h1
    xb1, yb1, xb2, yb2 = x2, y2, x2 + w2, y2 + h2

    inter_w = max(0.0, min(xa2, xb2) - max(xa1, xb1))
    inter_h = max(0.0, min(ya2, yb2) - max(ya1, yb1))
    inter = inter_w * inter_h

    area_a = max(0.0, w1) * max(0.0, h1)
    area_b = max(0.0, w2) * max(0.0, h2)
    union = area_a + area_b - inter
    if union <= 0.0:
        return 0.0
    return inter / union

def compute_iou_matrix(boxes1: torch.Tensor, boxes2: torch.Tensor) ->
torch.Tensor:
    if boxes1.ndim != 2 or boxes2.ndim != 2 or boxes1.size(1) != 4 or
boxes2.size(1) != 4:
        raise ValueError("boxes1 and boxes2 must be shape (N,4) and (M,4)")

    b1 = boxes1.float()
    b2 = boxes2.float()

    b1_x1 = b1[:, 0:1]
    b1_y1 = b1[:, 1:2]
    b1_x2 = b1[:, 0:1] + b1[:, 2:3]
    b1_y2 = b1[:, 1:2] + b1[:, 3:4]

    b2_x1 = b2[:, 0].unsqueeze(0)
    b2_y1 = b2[:, 1].unsqueeze(0)
    b2_x2 = (b2[:, 0] + b2[:, 2]).unsqueeze(0)
    b2_y2 = (b2[:, 1] + b2[:, 3]).unsqueeze(0)

    inter_w = torch.clamp(torch.minimum(b1_x2, b2_x2) - torch.maximum(b1_x1,
b2_x1), min=0.0)
```

```

inter_h = torch.clamp(torch.minimum(b1_y2, b2_y2) - torch.maximum(b1_y1,
b2_y1), min=0.0)
inter = inter_w * inter_h

area1 = torch.clamp(b1[:, 2:3], min=0.0) * torch.clamp(b1[:, 3:4], min=0.0)
area2 = torch.clamp(b2[:, 2], min=0.0).unsqueeze(0) * torch.clamp(b2[:, 3],
min=0.0).unsqueeze(0)
union = area1 + area2 - inter
return torch.where(union > 0.0, inter / union, torch.zeros_like(union))

```

3(h) Output: IoU test cases

- non-overlapping: 0.0
- identical: 1.0
- partial overlap: 0.142857...
- matrix example:
 - $\begin{bmatrix} 1.0 & 0.0 \\ 0.142857... & 0.0 \end{bmatrix}$

These outputs match the expected sanity cases exactly (no overlap, full overlap, partial overlap, and batched IoU).

4) Task 4: YOLO Network for LVIS

4(a) YOLO parameter choices and explanation

Chosen values:

- image size = 256
- grid size = 8×8 so $C = 64$
- anchors per cell $A = 5$ with ratios $[1/5, 1/3, 1, 3, 5]$
- vector length $V = 1 + 3 + 4 = 8$

Therefore the flattened output shape is $(B, C * A * V) = (B, 2560)$.

I chose an 8×8 grid because it gives finer spatial responsibility than 4×4 while keeping output size manageable for training from scratch.

4(b) Code: complete LVISYOLONet class

Code from `models.py`:

```

class LVISYOLONet(nn.Module):
    def __init__(self, num_classes: int = 3, num_cells: int = 16, num_anchors: int
= 5):
        super().__init__()
        grid = int(math.sqrt(num_cells))
        if grid * grid != num_cells:
            raise ValueError("num_cells must be a perfect square")
        self.num_classes = num_classes
        self.num_cells = num_cells

```

```

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

self.stem = nn.Sequential(
    nn.Conv2d(3, 32, kernel_size=3, stride=2, padding=1),
    nn.BatchNorm2d(32),
    nn.ReLU(inplace=False),
)
self.layer1 = SkipBlock(32, 64, downsample=True)
self.layer2 = SkipBlock(64, 128, downsample=True)
self.layer3 = SkipBlock(128, 256, downsample=True)
self.layer4 = SkipBlock(256, 256, downsample=False)

self.pool = nn.AdaptiveAvgPool2d((self.grid_size, self.grid_size))
self.head = nn.Conv2d(
    256,
    self.num_anchors * self.yolo_vec_len,
    kernel_size=1,
    stride=1,
    padding=0,
)

def forward(self, x: torch.Tensor) -> torch.Tensor:
    x = self.stem(x)
    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)
    x = self.layer4(x)
    x = self.pool(x)
    x = self.head(x)
    x = x.permute(0, 2, 3, 1).contiguous()
    x = x.view(x.shape[0], -1)
    return x

def count_parameters(self) -> int:
    return sum(p.numel() for p in self.parameters() if p.requires_grad)

```

This network uses a from-scratch skip-block backbone (no pretrained backbone), consistent with the homework constraints [4].

4(c) Total learnable parameters

LVISYOLONet parameters: **2,400,680**.

This parameter count is large enough to learn non-trivial features, but still feasible to train on the homework subset without pretrained backbones.

4(d) Code: custom dataloader and cell/anchor assignment logic

Code from `dataset.py`:

```

def assign_cell_and_anchor(bbox, img_size, grid_size, anchor_ratios):
    x, y, w, h = bbox
    cx = x + w / 2.0
    cy = y + h / 2.0

    cell_size = img_size / grid_size
    col = min(int(cx / cell_size), grid_size - 1)
    row = min(int(cy / cell_size), 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

    best_anchor = 0
    best_iou = -1.0
    best_aw = cell_size
    best_ah = cell_size
    anchor_area = cell_size * cell_size
    for a_idx, ratio in enumerate(anchor_ratios):
        a_h = float((anchor_area * ratio) ** 0.5)
        a_w = float(a_h / ratio)
        a_box = [cell_cx - a_w / 2.0, cell_cy - a_h / 2.0, a_w, a_h]
        iou = compute_iou(bbox, a_box)
        if iou > best_iou:
            best_iou = iou
            best_anchor = a_idx
            best_aw, best_ah = a_w, a_h

    sigma_w = w / best_aw if best_aw > 0 else 1.0
    sigma_h = h / best_ah if best_ah > 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"])
        bbox = ann["bbox"]
        cell_idx, anchor_idx, offsets = assign_cell_and_anchor(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 LVISDetectionYOLODataset(Dataset):
    def __init__(self, root_dir, split, num_classes=3, img_size=256, grid_size=4,
        anchor_ratios=(1/5,1/3,1,3,5)):
        self.img_dir = os.path.join(root_dir, split, "images")
        self.ann_dir = os.path.join(root_dir, split, "det_ann")
        self.img_size = img_size
        self.grid_size = grid_size
        self.anchor_ratios = list(anchor_ratios)
        self.num_classes = num_classes
        self.files = sorted([f for f in os.listdir(self.img_dir) if
            f.endswith(".jpg")])
        self.tf = transforms.Compose([transforms.Resize((img_size, img_size)),
            transforms.ToTensor()])

    def __getitem__(self, idx):
        fname = self.files[idx]
        img_path = os.path.join(self.img_dir, fname)
        ann_path = os.path.join(self.ann_dir, fname.replace(".jpg", ".json"))

        img = Image.open(img_path).convert("RGB")
        x = self.tf(img)
        with open(ann_path, "r", encoding="utf-8") as f:
            anns = json.load(f)
        gt_tensor = build_gt_yolo_tensor(anns, self.img_size, self.grid_size,
            self.anchor_ratios, self.num_classes)
        return {"image": x, "gt": gt_tensor, "anns": anns, "image_path": img_path,
            "file_name": fname}

```

Each GT object is assigned to one cell-anchor slot by center location + best-anchor IoU, then encoded as objectness + one-hot class + normalized offsets.

4(e) Code: three-loss training loop

Code from `train_yolo.py`:

```

def _compute_losses(pred, gt, num_classes, obj_scale, noobj_scale, max_pos_weight,
    class_weights):
    b = pred.shape[0]
    vec_len = 1 + num_classes + 4
    pred = pred.view(b, -1, vec_len)
    gt = gt.view(b, -1, vec_len)

    obj_pred = pred[:, :, 0]
    obj_gt = gt[:, :, 0]
    pos = torch.sum(obj_gt)
    neg = obj_gt.numel() - pos
    dyn_pos_w = torch.clamp(neg / (pos + 1e-6), min=1.0, max=max_pos_weight)
    pos_weight = torch.tensor(dyn_pos_w.item(), device=pred.device)
    bce_raw = F.binary_cross_entropy_with_logits(obj_pred, obj_gt,
        pos_weight=pos_weight, reduction="none")

```

```

pos_mask = obj_gt > 0.5
neg_mask = ~pos_mask
obj_pos = bce_raw[pos_mask].mean() if pos_mask.any() else torch.tensor(0.0,
device=pred.device)
obj_neg = bce_raw[neg_mask].mean() if neg_mask.any() else torch.tensor(0.0,
device=pred.device)
bce = obj_scale * obj_pos + noobj_scale * obj_neg

if pos_mask.any():
    cls_pred = pred[:, :, 1 : 1 + num_classes][pos_mask]
    cls_gt = torch.argmax(gt[:, :, 1 : 1 + num_classes][pos_mask], dim=1)
    ce = nn.CrossEntropyLoss(weight=class_weights)(cls_pred, cls_gt)
    box_pred = pred[:, :, 1 + num_classes : ][pos_mask]
    box_gt = gt[:, :, 1 + num_classes : ][pos_mask]
    mse = nn.MSELoss()(box_pred, box_gt)
else:
    ce = torch.tensor(0.0, device=pred.device)
    mse = torch.tensor(0.0, device=pred.device)

return bce, ce, mse

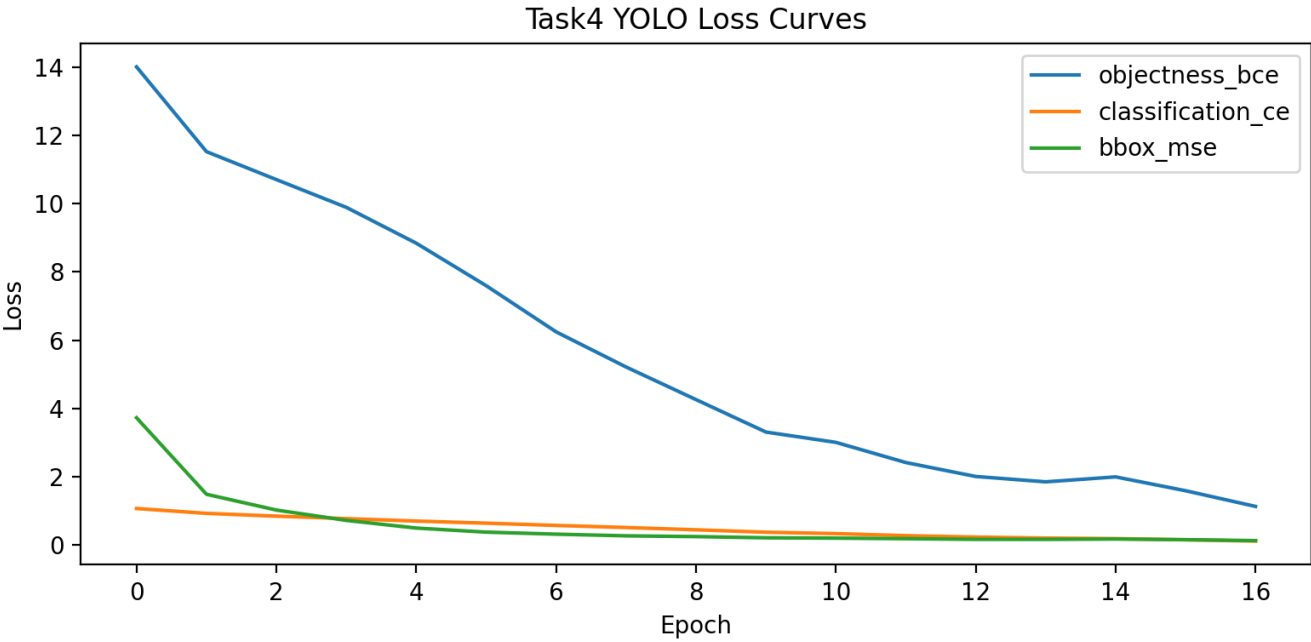
for batch in train_loader:
    x = batch["image"].to(device)
    gt = batch["gt"].to(device)
    optim.zero_grad()
    pred = model(x)
    bce, ce, mse = _compute_losses(
        pred,
        gt,
        cfg.num_classes,
        obj_scale=cfg.obj_scale,
        noobj_scale=cfg.noobj_scale,
        max_pos_weight=cfg.max_pos_weight,
        class_weights=class_w,
    )
    loss = bce + 2.0 * ce + 4.0 * mse
    loss.backward()
    optim.step()

```

The training objective uses BCE for objectness, CE for class prediction, and MSE for box offsets, with weighted combination to balance sparse positives vs dense negatives.

4(f) Figure: overlaid three losses

Figure 9. Task 4 loss curves (BCE/CE/MSE).



The overlaid curves show stable optimization without divergence, but the gap between classification and localization behavior remains visible.

4(g) Figure: 12 annotated validation images (8 good + 4 failure)

Figure 10. Curated Task 4 qualitative panel (8 good detections + 4 failures), each row GT vs prediction.

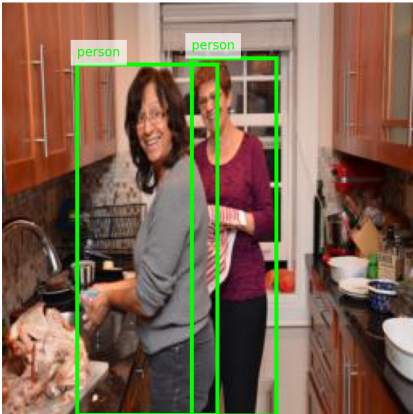




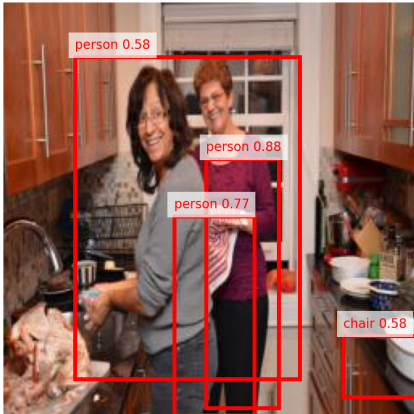
GOOD GT 000000089099.jpg (classes: person)



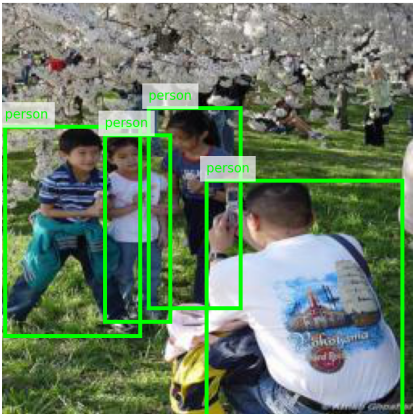
GOOD Pred tp/fp/fn=2/2/0 P=0.50 R=1.00 [works_well]



GOOD GT 000000557548.jpg (classes: person)



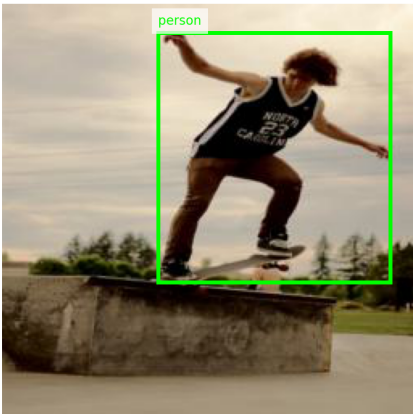
GOOD Pred tp/fp/fn=2/1/2 P=0.67 R=0.50 [works_well]



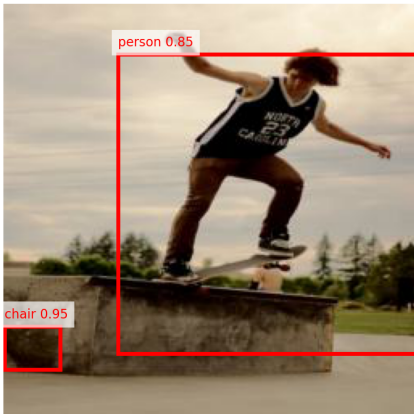
GOOD GT 000000120964.jpg (classes: person)



GOOD Pred tp/fp/fn=1/1/0 P=0.50 R=1.00 [works_well]



GOOD GT 000000185089.jpg (classes: person)



GOOD Pred tp/fp/fn=1/1/0 P=0.50 R=1.00 [works_well]



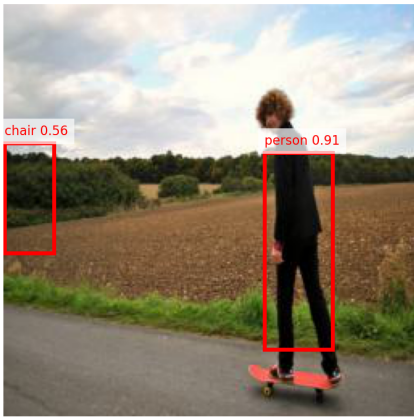
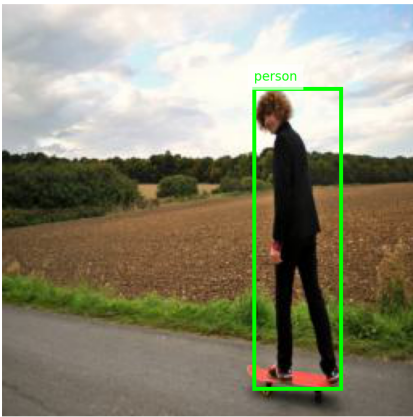
GOOD Pred tp/fp/fn=1/1/0 P=0.50 R=1.00 [works_well]



GOOD GT 000000350099.jpg (classes: person)

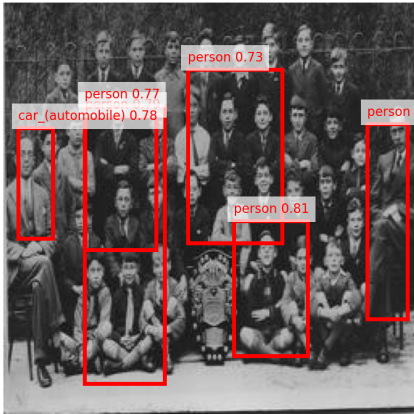


GOOD Pred tp/fp/fn=1/1/0 P=0.50 R=1.00 [works_well]



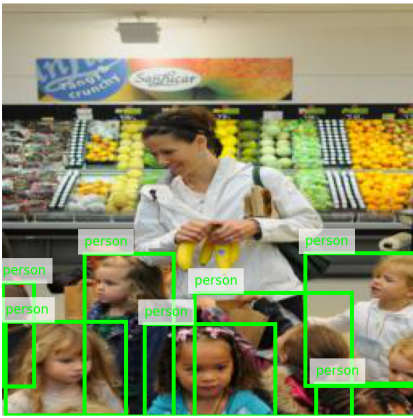
FAIL GT 000000072829.jpg (classes: person)

FAIL Pred tp/fp/fn=2/4/30 P=0.33 R=0.06 [missed_objects]



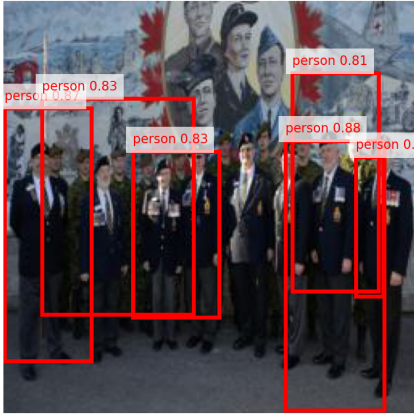
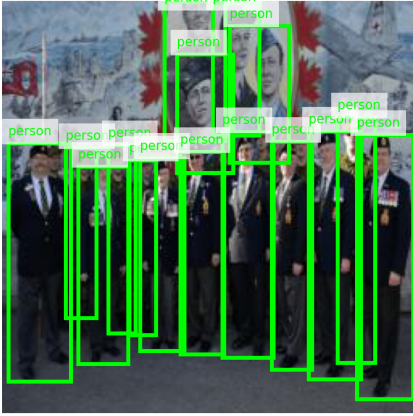
FAIL GT 000000040940.jpg (classes: person)

FAIL Pred tp/fp/fn=0/4/7 P=0.00 R=0.00 [wrong_labels]



FAIL GT 000000325157.jpg (classes: person)

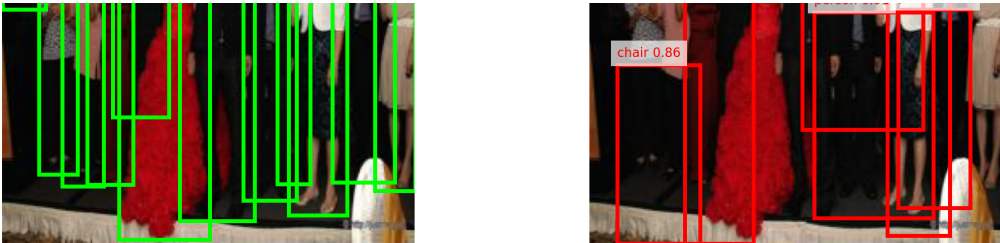
FAIL Pred tp/fp/fn=1/5/15 P=0.17 R=0.06 [poor_localization]



FAIL GT 000000354978.jpg (classes: person)

FAIL Pred tp/fp/fn=1/5/11 P=0.17 R=0.08 [false_positives]





- 8 good + 4 failure cases
 - good set covers all 3 classes
 - failure set includes missed objects, wrong labels, poor localization, and false positives
- This curation was done to show both capability and limitation, rather than only cherry-picked successes.

4(h) Performance discussion

Table 2. Final Task 4 per-class precision/recall at IoU >= 0.5 (threshold=0.5).

Class	Precision	Recall
car_(automobile)	0.0793	0.0826
chair	0.0270	0.0568
person	0.0734	0.1723

The biggest remaining weakness is class imbalance, especially for chair, and crowded scenes for person. person has the highest recall, but precision is still low because crowded images generate many hard negatives. chair is the hardest class in both precision and recall, which is consistent with small object scale and shape diversity. Overall detector quality is still limited.

5) Task 5: Semantic Segmentation on LVIS

5(a) Transpose convolution explanations

Matrix view: a 2D convolution (implemented as cross-correlation in deep learning libraries) can still be written as a linear map $y = W x$, where W is sparse and applies the kernel at shifted locations. Transpose convolution uses W^T , so each low-resolution input activation is spread back to a larger output grid with learnable weights. It is the transpose linear map, not an exact inverse [3].

Equation (3): transpose-convolution output size (height; width is analogous):

$$H_{out} = (H_{in} - 1) * stride - 2 * padding + dilation * (kernel_size - 1) + output_padding + 1$$

Here stride controls expansion, kernel_size controls receptive overlap, padding trims border growth, and output_padding resolves one-pixel shape ambiguity in strided decoding.

For the case in this question ($H_{in}=4$, $kernel=3$, $stride=2$, $padding=1$, $dilation=1$, $output_padding=0$):

$$H_{out} = (4 - 1)*2 - 2*1 + 1*(3 - 1) + 0 + 1 = 7.$$

ConvTranspose2d vs Upsample + Conv2d:

- **ConvTranspose2d**: upsampling and filtering are learned together in one layer; this is what the decoder uses in **LVISegNet** (`models.py`, `DecoderBlock`).
- **Upsample + Conv2d**: upsampling is fixed interpolation (nearest/bilinear), and only the following convolution is learned.

In practice, **Upsample + Conv2d** often reduces checkerboard risk because interpolation is fixed before learned filtering, while transpose convolution can show checkerboard artifacts if kernel/stride choices are poor.

mUnet encoder-decoder skip links vs ResNet skip links:

- Encoder-decoder skip links (U-Net/mUnet style) are long skips from encoder to decoder at matching resolution and are typically merged by concatenation; this explicitly restores edges and fine spatial detail.
- ResNet skip links are short intra-stage residual paths merged by addition ($x + F(x)$); their primary role is optimization and stable gradient flow, not decoder-side detail recovery [4].

This distinction matches the Task 5 outputs: region-level masks are reasonable, while small objects and boundaries are still the hardest cases.

5(b) Code: script used to run mUnet warm-up

Code snippet from `task5_munet_warmup.py`:

```
dls = DLStudio(
    dataroot="data/PurdueShapes5MultiObject/",
    image_size=[64, 64],
    path_saved_model=os.path.join(OUT_DIR, "munet_warmup.pt"),
    learning_rate=1e-4,
    epochs=epochs,
    batch_size=4,
    classes=("rectangle", "triangle", "disk", "oval", "star"),
    use_gpu=torch.cuda.is_available(),
)
seg = DLStudio.SemanticSegmentation(dl_studio=dls, max_num_objects=5)
train_ds = DLStudio.SemanticSegmentation.PurdueShapes5MultiObjectDataset(...)
test_ds = DLStudio.SemanticSegmentation.PurdueShapes5MultiObjectDataset(...)
seg.load_PurdueShapes5MultiObject_dataset(train_ds, test_ds)
model = seg.mUnet(skip_connections=True, depth=16)
seg.run_code_for_training_for_semantic_segmentation(model)
```

I used this warm-up to verify the encoder-decoder segmentation workflow before training the custom LVISegNet.

5(c) Figure: mUnet loss and ≥ 4 test triplets

Figure 11. mUnet warm-up training loss.

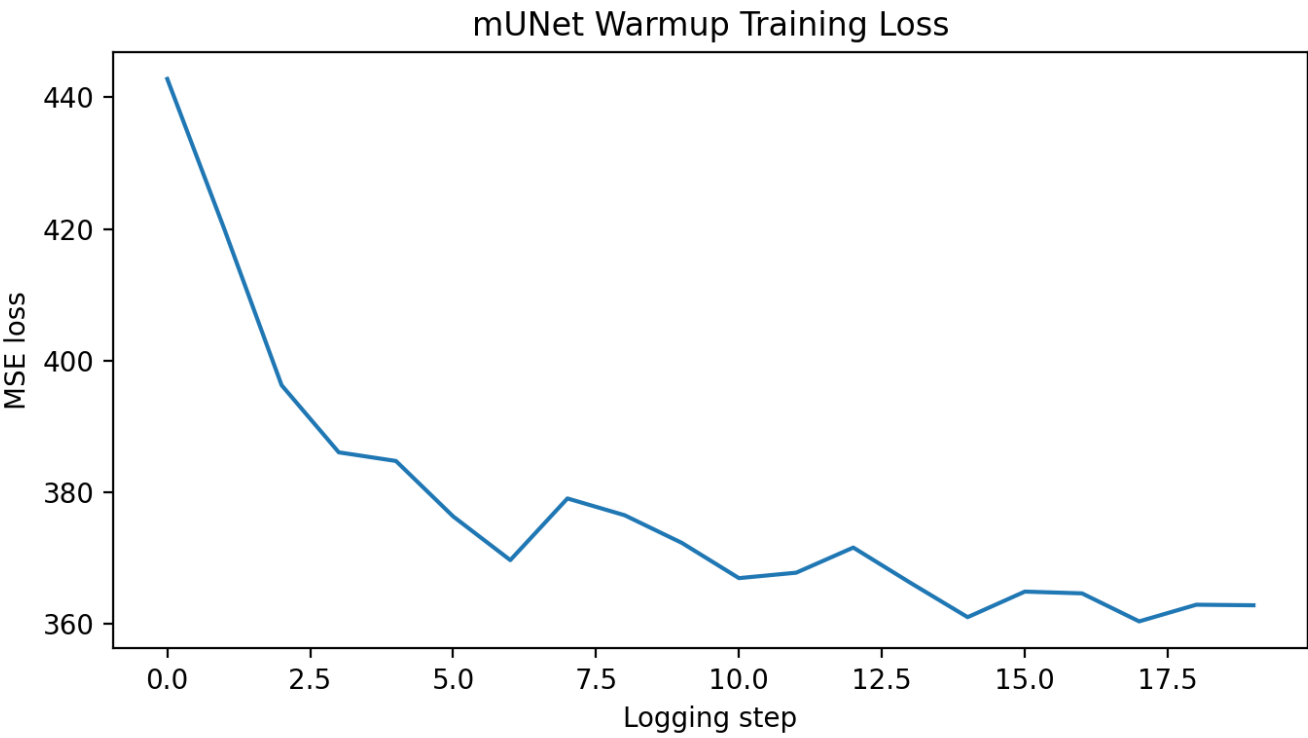
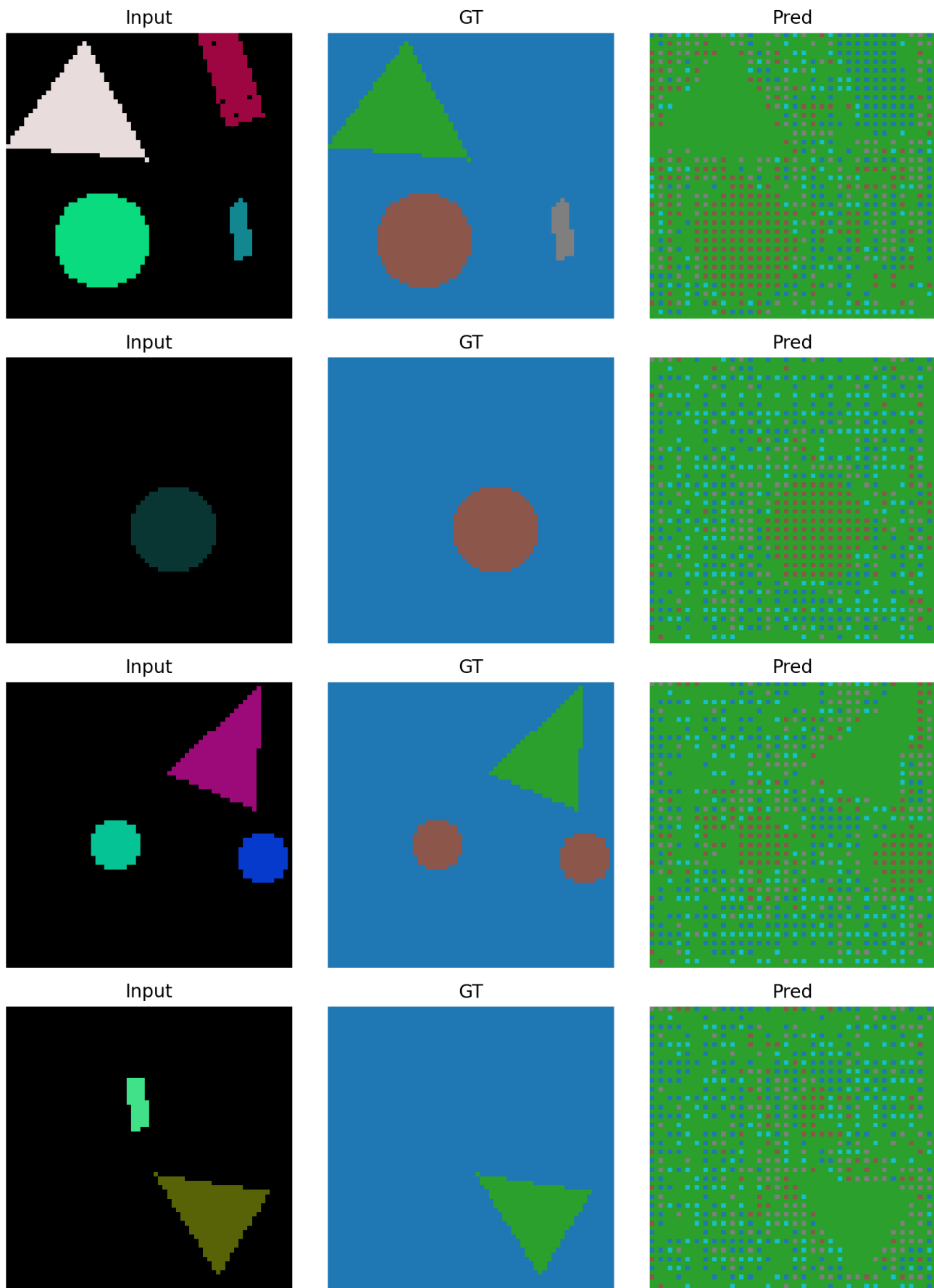


Figure 12. mUnet warm-up examples (input/GT/pred, >=4 images).



The warm-up confirms the model learns object masks on a simpler dataset, so the later LVIS errors are mainly due to data complexity and imbalance, not pipeline failure.

5(d) Code: complete `LVISSegNet` class

Code from `models.py`:

```
class LVISSegNet(nn.Module):
    def __init__(self, num_classes: int = 4):
        super().__init__()
        self.enc1 = EncoderBlock(3, 64)
        self.enc2 = EncoderBlock(64, 128)
        self.enc3 = EncoderBlock(128, 256)
        self.enc4 = EncoderBlock(256, 512)

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

        self.dec4 = DecoderBlock(512, 512, 256)
        self.dec3 = DecoderBlock(256, 256, 128)
        self.dec2 = DecoderBlock(128, 128, 64)
        self.dec1 = DecoderBlock(64, 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.dec4(x, s4)
        x = self.dec3(x, s3)
        x = self.dec2(x, s2)
        x = self.dec1(x, s1)
        return self.final(x)

    def count_parameters(self) -> int:
        return sum(p.numel() for p in self.parameters() if p.requires_grad)
```

The architecture follows a standard encoder-bottleneck-decoder design, where skip features (`s1..s4`) are reused during decoding to recover spatial detail.

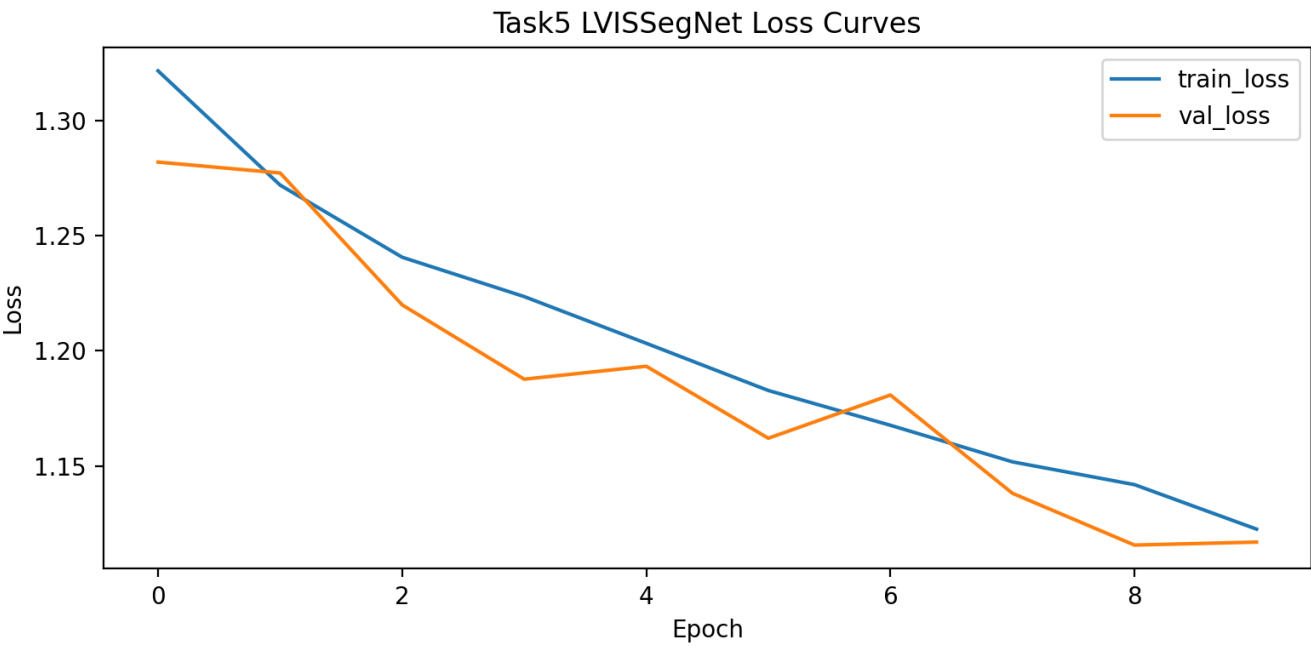
This is an encoder-decoder adaptation inspired by mUnet/U-Net style skip fusion [1,3,6].

5(e) Total learnable parameters

`LVISSegNet` parameters: **13,499,588**.

5(f) Figure: train/val loss curves for LVISSegNet

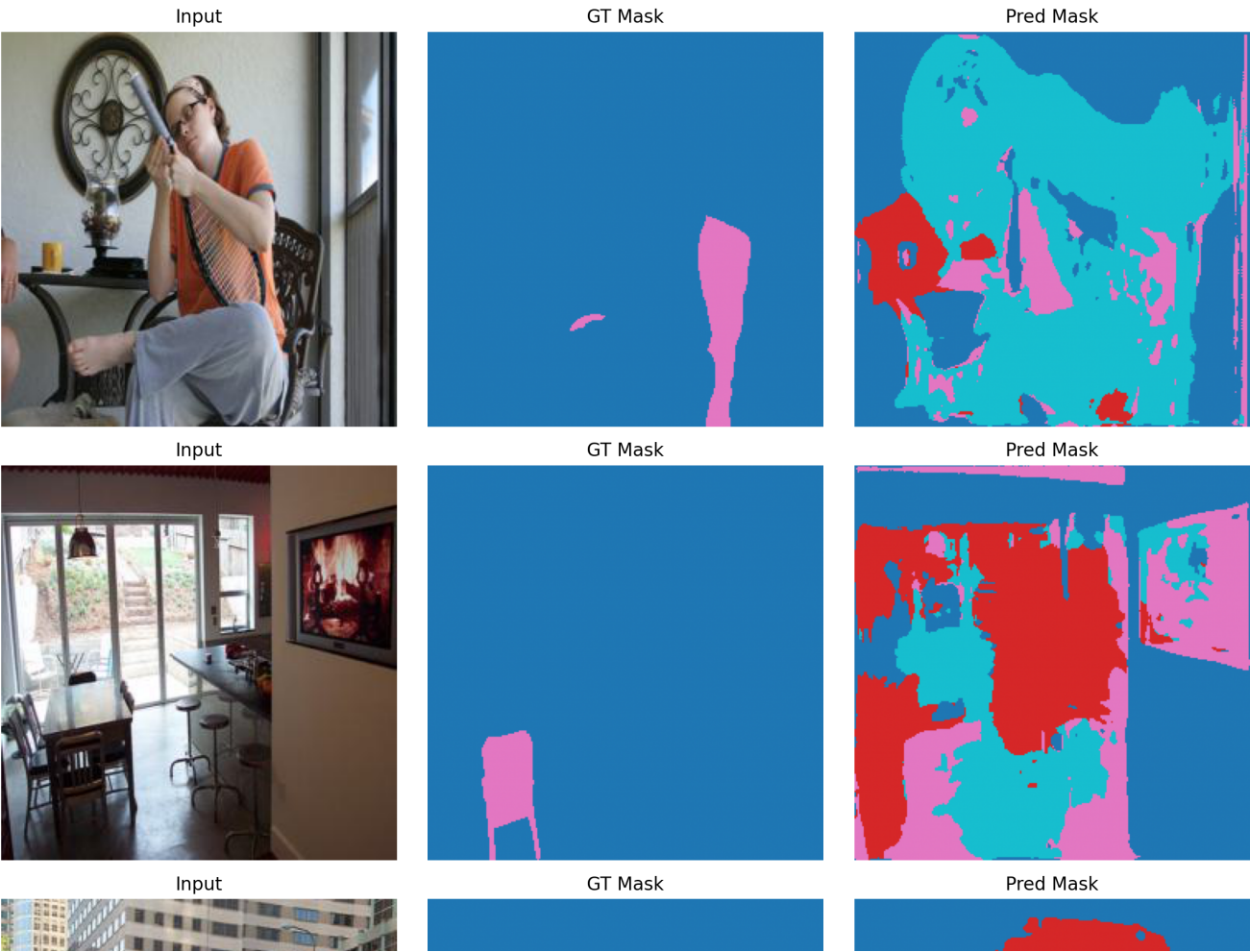
Figure 13. LVISSegNet train and validation loss.



Training and validation losses both decrease, but the validation curve stays higher, indicating limited generalization under class imbalance.

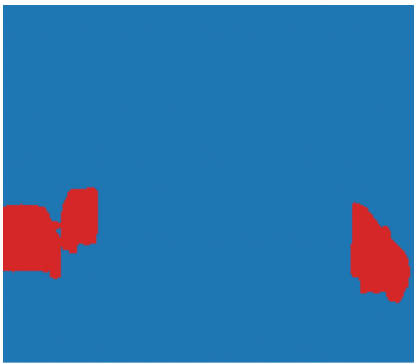
5(g) Figure: at least 8 validation triplets

Figure 14. LVISSegNet validation examples (input/GT/pred, 8 images).





Input



GT Mask



Pred Mask



Input



GT Mask



Pred Mask



Input



GT Mask



Pred Mask



Input



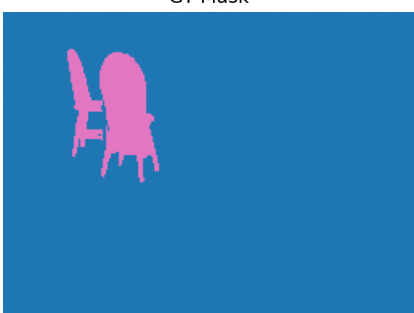
GT Mask



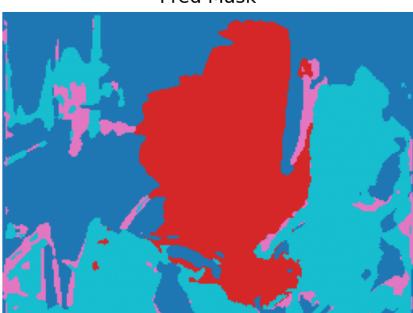
Pred Mask



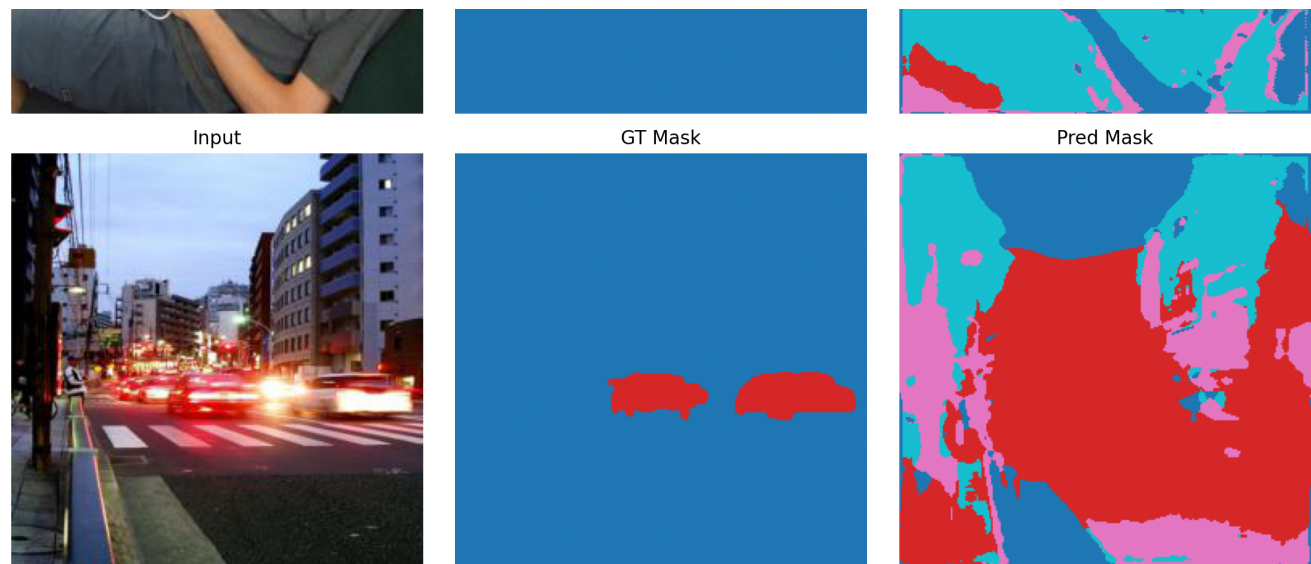
Input



GT Mask



Pred Mask



These examples are consistent with the quantitative metrics: coarse regions are often correct, but boundaries and small structures are less reliable.

5(h) Table: per-class pixel accuracy, overall pixel accuracy, mIoU

Table 3. Segmentation metrics

Metric	Value
Pixel Acc (background)	0.4134
Pixel Acc (car)	0.6804
Pixel Acc (chair)	0.4199
Pixel Acc (person)	0.5430
Overall pixel accuracy	0.4334
IoU (background)	0.4057
IoU (car)	0.1089
IoU (chair)	0.0547
IoU (person)	0.1617
mIoU (foreground classes only)	0.1085

Equation (4): per-class pixel accuracy.

$$\text{PixelAcc}_c = \text{correct_pixels}(c) / \text{gt_pixels}(c)$$

Equation (5): per-class IoU.

$$\text{IoU}_c = \text{TP}_c / (\text{TP}_c + \text{FP}_c + \text{FN}_c)$$

Even with moderate pixel accuracy, foreground IoU remains low because boundary errors and minority-class misses are penalized strongly in IoU.

5(i) Text: segmentation quality discussion

Foreground pixels are much rarer than background pixels, so class imbalance is still a major limitation. **person** has the best foreground IoU, while **chair** is the hardest class because of shape variability and clutter. Large regions are often segmented correctly, but small objects and boundaries remain noisy. The model is generally better at region-level localization than fine boundary recovery.

5(j) Code: YOLO precision/recall evaluation code

Code from **evaluate.py**:

```
def evaluate_detection_pr(
    gt_boxes,
    gt_labels,
    pred_boxes,
    pred_labels,
    pred_scores,
    num_classes,
    iou_thresh=0.5,
):
    metrics = {
        c: {"tp": 0.0, "fp": 0.0, "fn": 0.0, "precision": 0.0, "recall": 0.0}
        for c in range(num_classes)
    }

    for img_idx in range(len(gt_boxes)):
        gt_b = gt_boxes[img_idx]
        gt_l = gt_labels[img_idx]
        pd_b = pred_boxes[img_idx]
        pd_l = pred_labels[img_idx]
        pd_s = pred_scores[img_idx]

        order = sorted(range(len(pd_b)), key=lambda i: pd_s[i], reverse=True)

        for c in range(num_classes):
            gt_idx = [i for i, lb in enumerate(gt_l) if lb == c]
            pd_idx = [i for i in order if pd_l[i] == c]
            matched = set()

            for pi in pd_idx:
                best_iou = -1.0
                best_gi = None
                for gi in gt_idx:
                    if gi in matched:
                        continue
                    iou = compute_iou(pd_b[pi], gt_b[gi])
                    if iou > best_iou:
                        best_iou = iou
                        best_gi = gi
                if best_gi is not None and best_iou >= iou_thresh:
                    metrics[c]["tp"] += 1.0
                    matched.add(best_gi)
```

```
        else:
            metrics[c]["fp"] += 1.0

        metrics[c]["fn"] += float(len(gt_idx) - len(matched))

    for c in range(num_classes):
        tp = metrics[c]["tp"]
        fp = metrics[c]["fp"]
        fn = metrics[c]["fn"]
        metrics[c]["precision"] = tp / (tp + fp) if tp + fp > 0 else 0.0
        metrics[c]["recall"] = tp / (tp + fn) if tp + fn > 0 else 0.0

    return metrics
```

This evaluator uses class-wise greedy matching by confidence and IoU threshold, which is consistent with the standard precision/recall definition used in detection benchmarks.

5(k) Table: per-class precision/recall at IoU >= 0.5

Table 4. Detection precision/recall ([artifacts/task4/task4_precision_recall.json](#)).

Class	Precision	Recall
car_(automobile)	0.0793	0.0826
chair	0.0270	0.0568
person	0.0734	0.1723

Equation (6):

$Precision = TP / (TP + FP), Recall = TP / (TP + FN)$

The numbers indicate recall is generally higher than precision, so false positives remain the main bottleneck in the current detector.

5(l) Table: master summary (5 required rows)

Table 5. Master summary across required models.

Network	Dataset	Parameters	Key Metric
LOADnet2	PurdueShapes5	built-in	Test classification accuracy = 100.00%
NetForYOLO	PurdueDrEvalMulti	built-in	Overall accuracy = 90.0%
LVISYOLONet	LVIS (3 classes)	2,400,680	Per-class P/R at IoU>=0.5
mUnet	PurdueShapes5MultiObject	built-in	Warm-up segmentation examples + loss
LVISSegNet	LVIS (3 classes + background)	13,499,588	Foreground mIoU = 0.1085

This table summarizes the progression from warm-up models to LVIS models and highlights the performance gap between toy data and real-scene complexity.

5(m) Overall discussion and takeaways

The main challenge in moving from classification to detection is assignment and calibration. In classification there is one label per image, but in detection each image has a variable number of objects. The training pipeline must assign each object to the correct cell-anchor pair, and small assignment/decoding errors directly reduce precision and recall.

The three YOLO losses are tightly coupled: objectness controls candidate density, classification controls semantic correctness, and box regression controls localization quality. In these runs, objectness calibration was the main source of false positives. Rebalancing objectness and tightening decode filters improved stability, but recall for difficult categories remains limited.

Using the same LVIS split for detection and segmentation highlighted complementary behavior. Detection provides instance-level localization and category identity, but is sensitive to crowded scenes and small objects. Segmentation gives better region-level coverage, but boundary quality and minority-class performance are still constrained by class imbalance.

6) Bonus: DIoU Loss for Bounding Box Regression

6(a) Code: DIoU loss implementation

Equation (7):

$$L_{DIoU} = 1 - IoU + (\rho^2(b, b_{gt}) / c^2)$$

Compared with plain MSE on offsets, DIoU explicitly penalizes both overlap mismatch and center-distance mismatch.

```
def diou_loss_xywh(pred_boxes: torch.Tensor, gt_boxes: torch.Tensor) ->
torch.Tensor:
    px1 = pred_boxes[:, 0]
    py1 = pred_boxes[:, 1]
    px2 = pred_boxes[:, 0] + pred_boxes[:, 2]
    py2 = pred_boxes[:, 1] + pred_boxes[:, 3]

    gx1 = gt_boxes[:, 0]
    gy1 = gt_boxes[:, 1]
    gx2 = gt_boxes[:, 0] + gt_boxes[:, 2]
    gy2 = gt_boxes[:, 1] + gt_boxes[:, 3]

    inter_w = torch.clamp(torch.minimum(px2, gx2) - torch.maximum(px1, gx1),
min=0.0)
    inter_h = torch.clamp(torch.minimum(py2, gy2) - torch.maximum(py1, gy1),
min=0.0)
    inter = inter_w * inter_h

    area_p = torch.clamp(pred_boxes[:, 2], min=0.0) * torch.clamp(pred_boxes[:,
3], min=0.0)
    area_g = torch.clamp(gt_boxes[:, 2], min=0.0) * torch.clamp(gt_boxes[:, 3],
min=0.0)
    union = area_p + area_g - inter + 1e-9
```

```

iou = inter / union

pcx = pred_boxes[:, 0] + pred_boxes[:, 2] / 2.0
pcy = pred_boxes[:, 1] + pred_boxes[:, 3] / 2.0
gcx = gt_boxes[:, 0] + gt_boxes[:, 2] / 2.0
gcy = gt_boxes[:, 1] + gt_boxes[:, 3] / 2.0
rho2 = (pcx - gcx) ** 2 + (pcy - gcy) ** 2

ex1 = torch.minimum(px1, gx1)
ey1 = torch.minimum(py1, gy1)
ex2 = torch.maximum(px2, gx2)
ey2 = torch.maximum(py2, gy2)
c2 = (ex2 - ex1) ** 2 + (ey2 - ey1) ** 2 + 1e-9

return (1.0 - iou + rho2 / c2).mean()

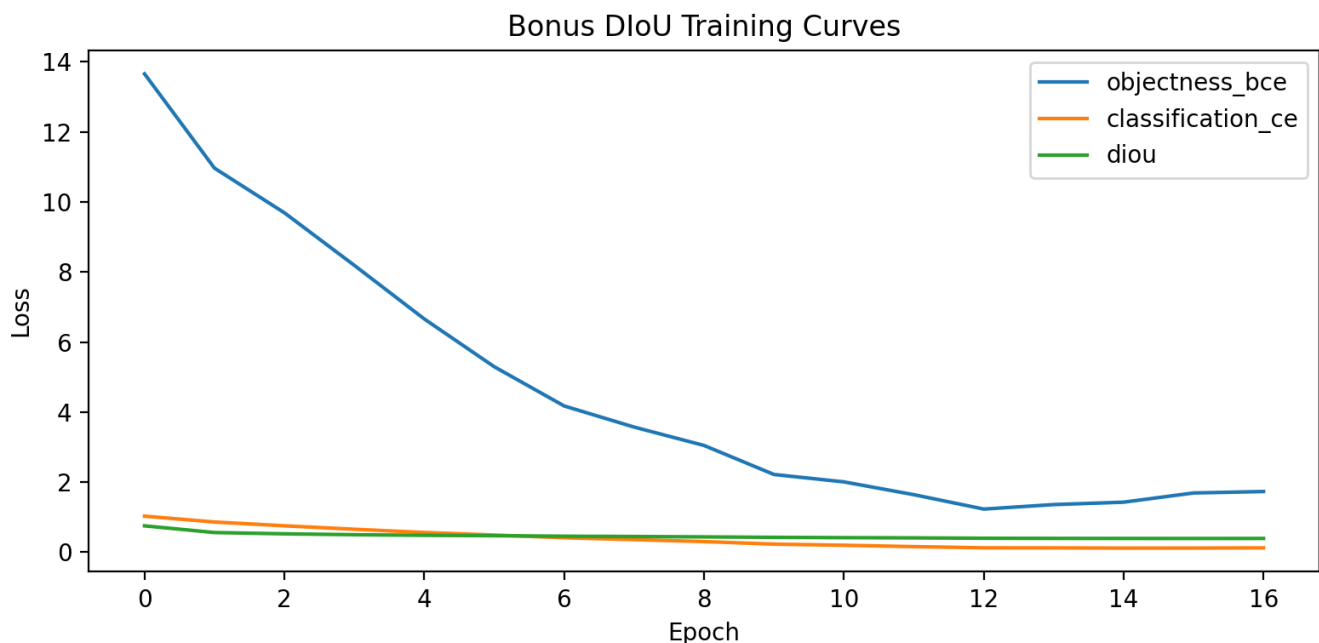
```

In the bonus training loop, DIoU replaces the MSE term while objectness and class losses are kept:

```
loss = l_obj + 2.0 * l_cls + 4.0 * l_diou
```

6(b) Figure: training loss curves for DIoU variant

Figure 15. Bonus DIoU training losses.



The DIoU run converges stably, so the comparison in Table 6 reflects objective differences rather than training instability.

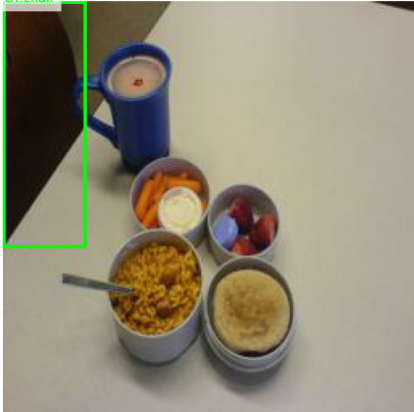
6(c) Figure: 12 annotated validation images for DIoU variant

Figure 16. Bonus DIoU validation panel (12 images, GT vs prediction).

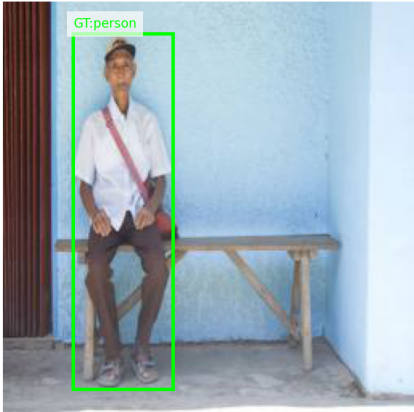
GOOD GT 000000312153.jpg (classes: car_(automobile)) GOOD DIoU tp/fp/fn=2/4/0 P=0.33 R=1.00 [works_well]



GOOD GT 000000115866.jpg (classes: chair)



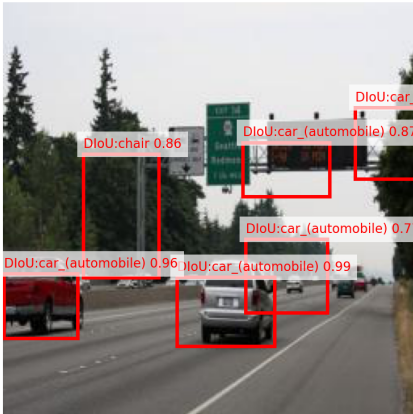
GOOD GT 000000075595.jpg (classes: person)



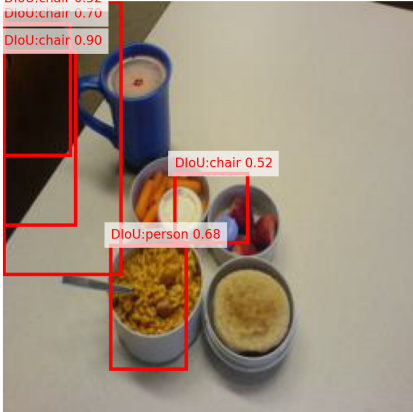
GOOD GT 000000555066.jpg (classes: car_(automobile))



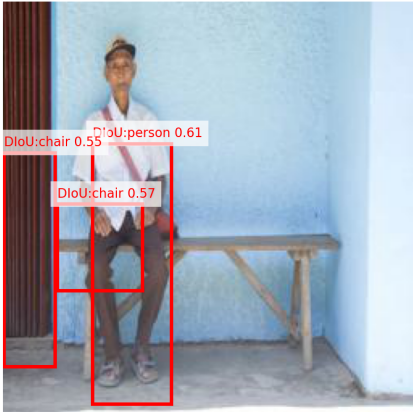
GOOD GT 000000114068.jpg (classes: car_(automobile))



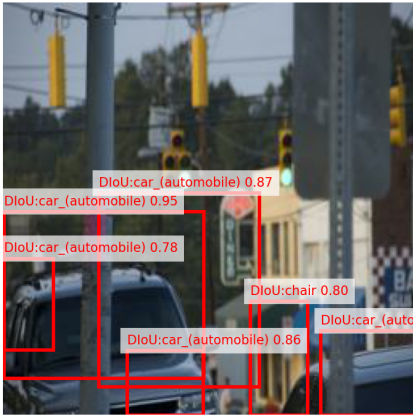
GOOD DIoU tp/fp/fn=1/4/0 P=0.20 R=1.00 [works_well]



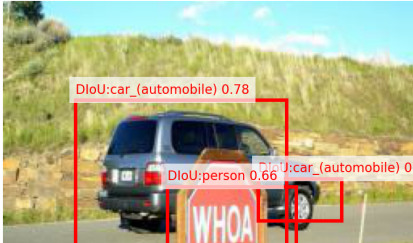
GOOD DIoU tp/fp/fn=1/2/0 P=0.33 R=1.00 [works_well]



GOOD DIoU tp/fp/fn=2/4/0 P=0.33 R=1.00 [works_well]

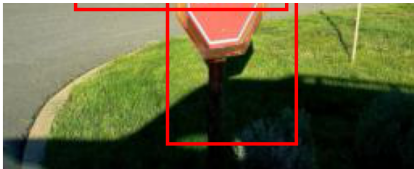


GOOD DIoU tp/fp/fn=1/2/0 P=0.33 R=1.00 [works_well]





GOOD GT 000000220666.jpg (classes: car_(automobile))



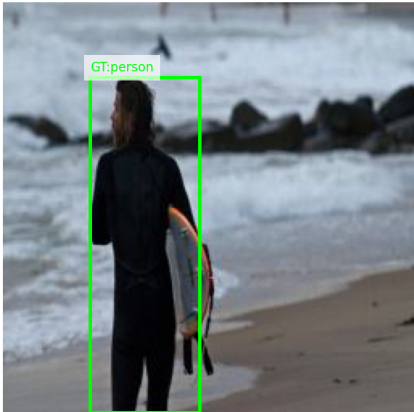
GOOD DIoU tp/fp/fn=1/2/0 P=0.33 R=1.00 [works_well]



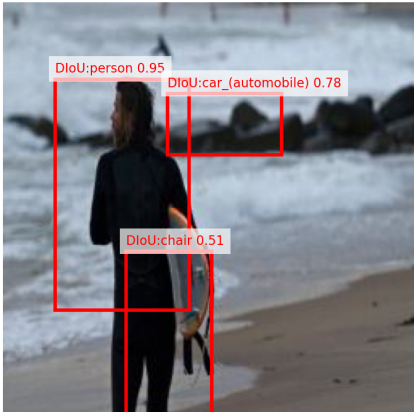
GOOD GT 000000240287.jpg (classes: person)



GOOD DIoU tp/fp/fn=1/2/0 P=0.33 R=1.00 [works_well]



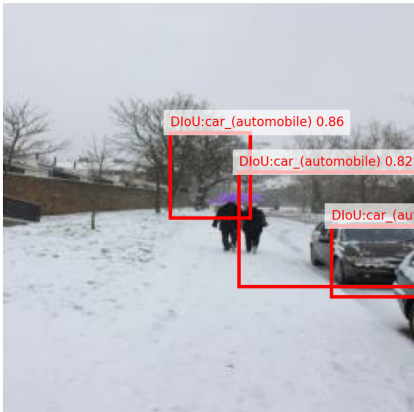
GOOD GT 000000352120.jpg (classes: car_(automobile))



GOOD DIoU tp/fp/fn=1/2/0 P=0.33 R=1.00 [works_well]



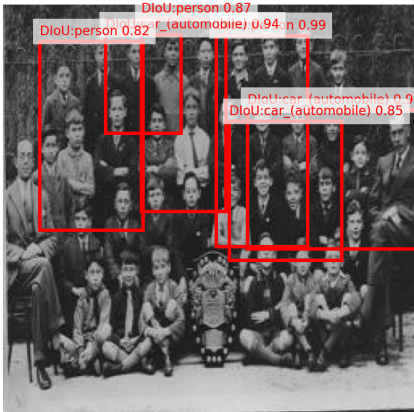
FAIL GT 000000072829.jpg (classes: person)



FAIL DIoU tp/fp/fn=0/6/32 P=0.00 R=0.00 [missed_objects]



FAIL GT 000000040940.jpg (classes: person)



FAIL DIoU tp/fp/fn=0/6/7 P=0.00 R=0.00 [wrong_labels]





GOOD GT 000000555066.jpg



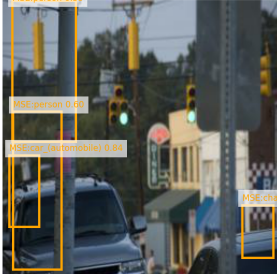
MSE tp/fp/fn=0/4/2



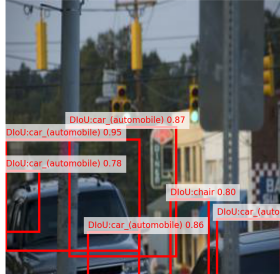
DloU tp/fp/fn=2/4/0 [works_well]



GOOD GT 000000114068.jpg



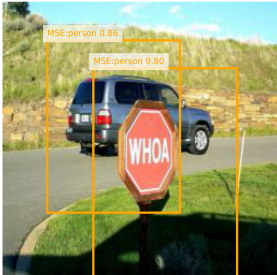
MSE tp/fp/fn=0/2/1



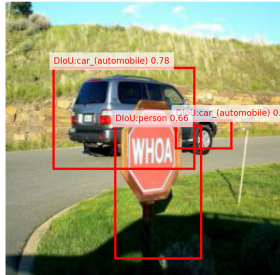
DloU tp/fp/fn=1/2/0 [works_well]



GOOD GT 000000220666.jpg



MSE tp/fp/fn=0/1/1



DloU tp/fp/fn=1/2/0 [works_well]



GOOD GT 000000240287.jpg



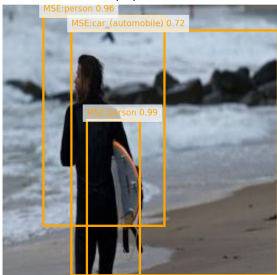
MSE tp/fp/fn=0/3/1



DloU tp/fp/fn=1/2/0 [works_well]



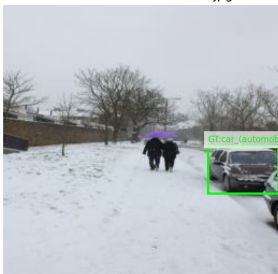
GOOD GT 000000352120.jpg



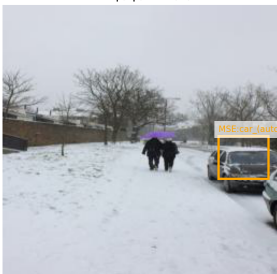
MSE tp/fp/fn=0/1/1



DloU tp/fp/fn=1/2/0 [works_well]



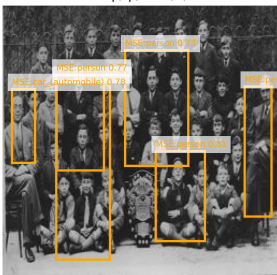
FAIL GT 000000072829.jpg



MSE tp/fp/fn=2/4/30



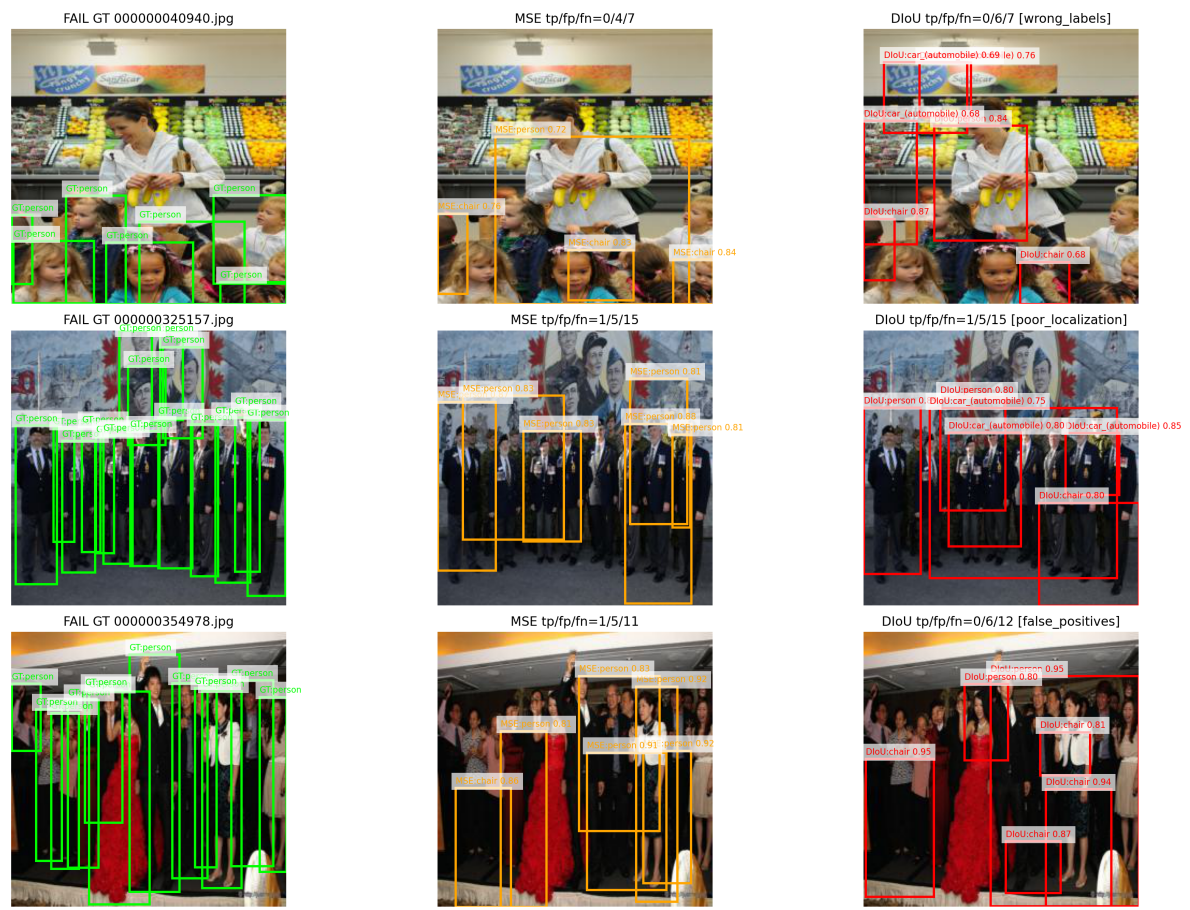
DloU tp/fp/fn=0/6/32 [missed_objects]



MSE tp/fp/fn=2/4/30



DloU tp/fp/fn=0/6/32 [missed_objects]



Showing MSE and DIoU side-by-side on the same images makes the precision-recall trade-off visually clear.

6(d) Table: MSE vs DIoU comparison at IoU >= 0.5

Table 6. Comparison from `artifacts/bonus/bonus_mse_vs_diou_comparison.json`.

Class	MSE Precision	MSE Recall	DIoU Precision	DIoU Recall
car_(automobile)	0.0793	0.0826	0.0337	0.1377
chair	0.0270	0.0568	0.0255	0.0720
person	0.0734	0.1723	0.0723	0.1052

DIoU mainly improves coverage (`car`, `chair` recall), while MSE remains stronger on precision calibration.

6(e) Discussion: DIoU vs MSE localization quality

Using the final implementation, DIoU and MSE show a clear trade-off at IoU >= 0.5. DIoU improves recall for `car` (0.1377 vs 0.0826) and `chair` (0.0720 vs 0.0568), but precision is lower for both classes (0.0337 vs 0.0793, 0.0255 vs 0.0270). For `person`, DIoU precision is similar to MSE (0.0723 vs 0.0734) while recall is lower (0.1052 vs 0.1723). Overall, in this training setup, DIoU favors coverage while MSE remains better calibrated for precision.

7) References

[1] A. Kak, DLStudio platform. <https://engineering.purdue.edu/kak/distDLS/>

- [2] A. Kak, YOLOLogic module. <https://engineering.purdue.edu/kak/distYOLO/>
- [3] A. Kak, Semantic segmentation lecture notes. <https://engineering.purdue.edu/kak/pdf-kak/SemanticSeg.pdf>
- [4] A. Kak, Skip connections and normalization notes. <https://engineering.purdue.edu/kak/pdf-kak/SkipConsAndBN.pdf>
- [5] J. Redmon et al., YOLO (CVPR 2016). <https://arxiv.org/abs/1506.02640>
- [6] O. Ronneberger et al., U-Net (MICCAI 2015). <https://arxiv.org/abs/1505.04597>