BME646 and ECE60146: Homework 6

Spring 2026
Due Date: Tuesday, March 10, 2026, 11:59 pm ET
TA: Somosmita Mitra (mitra26@purdue.edu)

Turn in typed solutions via Gradescope. Post questions to Piazza. Additional instructions can be found at the end. **Late submissions will be accepted with penalty: -10 points per late day, up to 5 days.**

This is a VERY challenging homework. It spans two weeks. Start early!

1 Introduction

In previous homeworks, you built classification networks of increasing sophistication: from simple CNNs (`LVISNet` in HW4) to deep residual networks with skip connections and normalization (`LVISSkipNet` in HW5). In all those cases, the network produced a single output, a class label for the entire image.

In this homework, you will tackle two fundamentally harder problems:

1. **Multi-instance object detection:** Given an image containing multiple objects of interest, predict *both* the class label *and* the bounding box coordinates for each object instance. This requires the network to make multiple simultaneous inferences from a single input.

2. **Semantic segmentation:** Assign a class label to *every pixel* in the image, producing a dense output map the same size as the input. This requires an encoder-decoder architecture with transpose convolutions.

As discussed in the lectures on multi-instance detection [2] and semantic segmentation [3], these tasks introduce several new concepts:

- **Dual-inferencing networks:** Networks that produce two different outputs (classification and bounding-box regression) for the same input, requiring two loss functions trained simultaneously.

- **YOLO logic:** Dividing the image into a grid of cells, associating anchor boxes with each cell, and using a structured output tensor (`yolo_tensor`) to predict objectness, class labels, and bounding box offsets.

- **Transpose convolutions:** Operations that "reverse" the spatial downsampling performed by standard convolutions, enabling the decoder in an encoder-decoder network to upsample back to the original image resolution.
- **Encoder-decoder architectures:** Networks like mUnet that first encode the image into high-level abstractions and then decode those abstractions back into pixel-level predictions, using skip connections between encoder and decoder to recover fine spatial detail.

Concretely, you will:

- Experiment with DLStudio's single-instance detector (`LOADnet2`) on PurdueShapes5 to understand dual-loss training
- Study the YOLO logic for multi-instance detection: image cells, anchor boxes, yolo_vectors, and yolo_tensors
- Build a multi-instance object detection dataset from LVIS with bounding box annotations
- Implement a YOLO-style network for multi-instance detection on your LVIS dataset, training with three loss functions (objectness, classification, and bounding box regression)
- Study encoder-decoder architectures and transpose convolutions by running DLStudio's mUnet on PurdueShapes5MultiObject
- Build a semantic segmentation dataset from LVIS polygon annotations for your same 3 categories, and train an encoder-decoder network to produce pixel-level segmentation masks
- Compare and contrast multi-instance detection and semantic segmentation *on the same LVIS data* in a final analysis

1.1 Why Multi-Instance Detection Is Hard

The fundamental challenge of multi-instance detection is the complexity of the input-output relationship [2]. For classification, the relationship is simple: one image in, one label out. For detection with localization, the output must include both a label and four bounding-box parameters for *each* object instance, and the number of instances varies across images.

The YOLO (You Only Look Once) approach [5] solves this by imposing spatial structure on the predictions: the image is divided into a grid of cells, each cell is assigned a set of anchor boxes with different aspect ratios, and the network outputs a structured tensor encoding objectness scores, class probabilities, and bounding-box offsets for every cell-anchor combination. The key insight is that **the cell whose center is closest to the center of an object is responsible for detecting that object**, and **the anchor box whose aspect ratio best matches the object is responsible for regressing its bounding box**.

1.2 Why Semantic Segmentation Requires a Different Architecture

Semantic segmentation demands pixel-level output: a class label for every pixel. Standard CNNs progressively downsample the spatial resolution through pooling and strided convolutions, ending with a compact feature vector suitable for classification but far too small for pixel-level prediction [3].

Encoder-decoder architectures address this by pairing the downsampling encoder with an upsampling decoder that uses **transpose convolutions** (`nn.ConvTranspose2d`) to progressively restore spatial resolution. Skip connections between encoder and decoder stages help recover fine spatial details lost during encoding. The U-Net architecture [6] pioneered this approach, and DLStudio's `mUnet` implements a multi-class variant.

1.3 Quick Reference: Lecture Slide Numbers

Use this table to look up the relevant lecture material for each topic in this homework.

HW6 Topic	Lecture	Slides
Multi-Instance Object Detection [2]		
Dual-inferencing CNN architecture	MultiDetection	11–18
Cross-entropy loss (deep dive)	MultiDetection	19–37
Regression loss (MSE for bounding boxes)	MultiDetection	38–49
PurdueShapes5 dataset	MultiDetection	50–56
LOADnet2 for single-instance detection	MultiDetection	60–73
PurdueDrEvalDataset (single-instance, harder)	MultiDetection	74–94
PurdueDrEvalMultiDataset (multi-instance)	MultiDetection	95–102
YOLO logic: cells, anchors, yolo_vector/tensor	MultiDetection	103–118
YOLOLogic module and NetForYOLO	MultiDetection	119–121
Training multi-instance detector (3 losses)	MultiDetection	122–130
Evaluating multi-instance detectors	MultiDetection	131–137
Testing and visualizing results	MultiDetection	138–153
Semantic Segmentation [3]		
Atrous (dilated) convolutions	SemanticSeg	9–16
Survey of segmentation networks (Gen 1–5)	SemanticSeg	17–28
Transpose convolutions	SemanticSeg	29–45
Kernel size, padding, output size for transpose conv	SemanticSeg	46–54
Stride and upsampling in transpose conv	SemanticSeg	55–62
Building blocks for mUnet	SemanticSeg	63–67
mUnet architecture	SemanticSeg	68–71
PurdueShapes5MultiObject dataset	SemanticSeg	72–80
Training mUnet	SemanticSeg	81–82
Testing mUnet on unseen data	SemanticSeg	83+

Which slides to read for each task:

Task	Key slides
Task 1: Single-instance detection	MultiDetection 11–18, 38–73
Task 2: YOLO logic study	MultiDetection 103–130
Task 3: LVIS detection dataset	MultiDetection 50–56, 74–80
Task 4: YOLO network for LVIS	MultiDetection 103–153
Task 5: Semantic segmentation	SemanticSeg 29–83
Task 5 (LVIS segmentation)	SemanticSeg 63–83

2 Setting Up Your Environment

2.1 Prerequisites from Previous Homeworks

This homework builds on your previous setup. Ensure you have:

- DLStudio version 2.5.5 installed (with the Examples directory)

- YOLOLogic module installed from https://engineering.purdue.edu/kak/distYOLO/
- LVIS annotations (`lvis_v1_train.json`) and COCO 2017 training images (from HW4/HW5)
- Your `train.py` and `evaluate.py` from previous homeworks
- PurdueShapes5 and PurdueShapes5MultiObject datasets (download from the DLStudio image datasets link)

2.2 Installing YOLOLogic

Download and install the YOLOLogic module from the distribution link:

https://engineering.purdue.edu/kak/distYOLO/

Verify installation by running:

```
from YOLOLogic import YOLOLogic
print("YOLOLogic imported successfully")
```

2.3 Downloading Required Datasets

Download the datasets archive from the DLStudio page:

`datasets_for_YOLO.tar.gz`

Uncompressing this archive will produce:

```
Purdue_Dr_Eval_Dataset-clutter-10-noise-20
    -size-10000-train.gz
Purdue_Dr_Eval_Multi_Dataset-clutter-10
    -noise-20-size-10000-train.gz
```

You will also need the `PurdueShapes5MultiObject` dataset for Task 5. This is part of the main DLStudio datasets download ("Download the image datasets for the main DLStudio module" on the DLStudio page).

2.4 Locating Key DLStudio/YOLOLogic Components

Familiarize yourself with:

1. `DLStudio/DLStudio.py`: The `DetectAndLocalize` inner class and its `LOADnet2` network

2. `DLStudio/DLStudio.py`: The `SemanticSegmentation` inner class and its `mUnet` network

3. `YOLOLogic/YOLOLogic.py`: The `YoloObjectDetector` inner class and its `NetForYOLO` network

4. Example scripts in the DLStudio and YOLOLogic Examples directories

Read the docstrings associated with each class definition. Understanding the structure of these classes is essential for this homework.

2.5 How to Use the LVIS Bounding Box Annotations

For this homework, you will need **bounding boxes** in addition to the class labels from LVIS. In previous homeworks, you used LVIS annotations only for classification (filtering images by category). Now you must also extract the bounding box coordinates for each object instance.

Before jumping into code, it is important to understand the structure of LVIS annotations. Each annotation in LVIS is a dictionary with the following fields:

```
{
    "id": 12345,              # unique annotation ID
    "image_id": 245915,       # integer ID for the image
    "category_id": 1,         # LVIS category ID
    "bbox": [86, 65, 220, 334],
        # bounding box: [top-left x, top-left y,
        #                width, height]
    "area": 73480.0,          # area in pixels
    "segmentation": [...],    # polygon vertices
        # (needed for Task 3 mask construction)
}
```

Note: The `bbox` field is in `[x, y, width, height]` format, where `(x, y)` is the top-left corner. This is the same format used by COCO annotations.

The following code shows how to access LVIS annotations, build a category mapping, retrieve images for your chosen categories, and display a sample image with bounding box annotations drawn. After importing the required modules, you can run this code and visually verify the output.

```
import json
import numpy as np
import matplotlib.pyplot as plt
```

```
import matplotlib.patches as patches
from PIL import Image
import os

# Load LVIS annotations
ann_file = 'lvis_v1_train.json'
with open(ann_file, 'r') as f:
    lvis_data = json.load(f)

# Your chosen categories
class_list = ['car_(automobile)', 'chair', 'bottle']

########################################
# Build mappings: LVIS cat ID -> index
########################################
# Find LVIS category IDs for your classes
cat_name_to_id = {}
for cat in lvis_data['categories']:
    if cat['name'] in class_list:
        cat_name_to_id[cat['name']] = cat['id']
print("Category name -> LVIS ID:",
      cat_name_to_id)

# Map LVIS cat IDs to your class indices
# (0, 1, 2)
lvis_id_to_idx = {}
for idx, name in enumerate(class_list):
    lvis_id_to_idx[cat_name_to_id[name]] = idx
print("LVIS ID -> class index:",
      lvis_id_to_idx)

########################################
# Build image_id -> annotations mapping
########################################
# Group annotations by image_id, keeping
# only annotations for your categories
from collections import defaultdict
img_to_anns = defaultdict(list)
target_cat_ids = set(lvis_id_to_idx.keys())
min_area = 2500  # 50x50 minimum

for ann in lvis_data['annotations']:
    if (ann['category_id'] in target_cat_ids
            and ann['area'] >= min_area):
        img_to_anns[ann['image_id']].append(ann)

# Keep only images that have at least one
# qualifying foreground object
valid_img_ids = [
```

```
    img_id for img_id, anns
    in img_to_anns.items() if len(anns) > 0
]
print(f"Found {len(valid_img_ids)} images "
      f"with qualifying objects")

########################################
# Build image_id -> file info mapping
########################################
img_id_to_info = {}
for img_info in lvis_data['images']:
    img_id_to_info[img_info['id']] = img_info

########################################
# Display one random image with
# bounding box annotations
########################################
idx = np.random.randint(0, len(valid_img_ids))
img_id = valid_img_ids[idx]
img_info = img_id_to_info[img_id]

# LVIS uses COCO images; construct path
# LVIS entries have 'coco_url' (not 'file_name')
coco_img_dir = 'path/to/coco/train2017'
fname = img_info['coco_url'].split('/')[-1]
img_path = os.path.join(
    coco_img_dir, fname)
image = Image.open(img_path).convert('RGB')
W_orig, H_orig = image.size

fig, ax = plt.subplots(1, 1, figsize=(8, 8))
ax.imshow(image)

colors = ['red', 'green', 'blue']
anns = img_to_anns[img_id]
for ann in anns:
    [x, y, w, h] = ann['bbox']
    cls_idx = lvis_id_to_idx[ann['category_id']]
    label = class_list[cls_idx]
    color = colors[cls_idx]

    rect = patches.Rectangle(
        (x, y), w, h,
        linewidth=2, edgecolor=color,
        facecolor='none')
    ax.add_patch(rect)
    ax.text(x, y - 5, label,
            fontsize=12, color=color,
            fontweight='bold',
```

```
            bbox=dict(facecolor='white',
                      alpha=0.7,
                      edgecolor='none'))

ax.set_axis_off()
ax.set_title(f"Image {img_id}: "
             f"{len(anns)} objects")
plt.tight_layout()
plt.show()
```

Important notes:

- Unlike `pycocotools`, the code above loads the raw JSON directly. You *may* also use the `lvis` Python package (`pip install lvis`) which provides a similar API to `pycocotools`. Either approach is acceptable.
- LVIS images are COCO images. LVIS image entries contain a `coco_url` field (e.g., `http://images.cocodataset.org/train2017/000000391895.jpg`) from which you can extract the filename. There is no `file_name` field in LVIS, use `img_info['coco_url'].split('/')[-1]` to get the COCO filename. You should already have these images from HW4.
- When resizing images to 256×256, remember to scale the bounding box coordinates proportionally (see Section 2.6).
- Some images may contain objects from categories *outside* your 3 chosen classes. Ignore those annotations, only process annotations whose `category_id` maps to one of your 3 classes.

2.6 LVIS Detection Dataset Requirements

For Task 3, you will build a multi-instance object **detection** dataset from LVIS. Unlike previous homeworks where you only needed class labels, you now also need **bounding box annotations** for each object instance.

2.6.1 Category Selection

Choose **3 LVIS categories** that satisfy all of the following:

- Each category must have at least **500 images** containing that category in the LVIS training split
- The categories should be visually distinct from one another

- At least **30%** of images per category must contain **multiple instances** of that category (or multiple foreground objects from your 3 categories)
- Each foreground object bounding box must have an area of at least $50 \times 50 = 2{,}500$ pixels (use the `area` field in the LVIS annotation)

Suggested categories (you may choose others): `['car_(automobile)', 'chair', 'bottle']`. Other viable sets include `['dog', 'cat', 'bird']` or `['cup', 'bowl', 'plate']`. **Important:** LVIS uses verbose category names (e.g., `'car_(automobile)'` not `'car'`). Browse `lvis_data['categories']` to find exact names and instance counts for your chosen categories.

2.6.2 Dataset Construction

1. Filter LVIS images so that each image contains at least one foreground object from your 3 categories with area $\geq 2{,}500$ pixels.
2. Resize all images to 256×256 pixels. **Scale bounding box coordinates proportionally** after resizing.
3. Use images from the LVIS **training split** for your training set and hold out 20% as a validation set (split before any augmentation).
4. Aim for approximately **3,000–5,000 training images** and **800–1,200 validation images**.

2.6.3 Bounding Box Format and Rescaling

LVIS bounding boxes are in `[x, y, width, height]` format (top-left corner). Store them in this format. When resizing from the original image dimensions $(W_{\text{orig}}, H_{\text{orig}})$ to $(256, 256)$, scale as:

$$x_{\text{new}} = x \cdot \frac{256}{W_{\text{orig}}}, \quad y_{\text{new}} = y \cdot \frac{256}{H_{\text{orig}}} \tag{1}$$

$$w_{\text{new}} = w \cdot \frac{256}{W_{\text{orig}}}, \quad h_{\text{new}} = h \cdot \frac{256}{H_{\text{orig}}} \tag{2}$$

Rescaling code template:

```
from PIL import Image

def resize_image_and_boxes(img_path, anns,
                           target_size=256):
```

```
    """Resize image and scale bounding boxes.
    Args:
        img_path: path to original image
        anns: list of annotation dicts with
              'bbox' and 'category_id' fields
        target_size: output image size
    Returns:
        resized_image: PIL Image
        scaled_anns: list of dicts with
                     scaled bbox values
    """
    image = Image.open(img_path).convert('RGB')
    W_orig, H_orig = image.size
    image = image.resize(
        (target_size, target_size))

    scale_x = target_size / W_orig
    scale_y = target_size / H_orig

    scaled_anns = []
    for ann in anns:
        x, y, w, h = ann['bbox']
        scaled_ann = {
            'category_id': ann['category_id'],
            'bbox': [
                x * scale_x,
                y * scale_y,
                w * scale_x,
                h * scale_y
            ]
        }
        scaled_anns.append(scaled_ann)
    return image, scaled_anns
```

Common mistake: Students often mix up width/height scaling when images are not square. An image that is 640×480 has $W_{\text{orig}} = 640$ and $H_{\text{orig}} = 480$. The x-coordinates and widths scale by $256/640$, while y-coordinates and heights scale by $256/480$. Getting this wrong will misalign your bounding boxes.

2.6.4 Dataset Verification Checklist

Before training any detector, verify and **report** the following:

1. Print the number of images in training and validation sets
2. Print the number of object instances per category in the training set

3. Print the average number of foreground objects per image

4. Display a 3×3 grid of sample images with bounding boxes and class labels drawn (3 images per category)

5. List your 3 LVIS category names

Include the output of this verification and the sample image grid in your report.

2.7 LVIS Segmentation Mask Preparation

In Task 5, you will also use your LVIS dataset for **semantic segmentation**. LVIS provides polygon segmentation annotations for every object instance. You will convert these polygons into pixel-level segmentation masks.

2.7.1 Understanding LVIS Segmentation Annotations

Each LVIS annotation includes a `segmentation` field containing one or more polygon outlines:

```
{
    "id": 12345,
    "image_id": 245915,
    "category_id": 1,
    "segmentation": [
        [x1, y1, x2, y2, ..., xN, yN]
    ],
    # Each sub-list is a polygon: a flat list
    # of (x, y) vertex coordinates
    "bbox": [86, 65, 220, 334],
    "area": 73480.0
}
```

The `segmentation` field is a list of polygons (some objects have multiple disconnected parts). Each polygon is a flat list of alternating x, y coordinates: `[x1, y1, x2, y2, ..., xN, yN]`.

2.7.2 Converting Polygons to Masks

The following code shows how to convert polygon annotations into a pixel-level segmentation mask for one image. The mask has the same spatial dimensions as the (resized) image, and each pixel is assigned the class index of the object it belongs to (or 0 for background).

```python
from PIL import Image, ImageDraw
import numpy as np

def create_segmentation_mask(anns, img_w, img_h,
        target_size, lvis_id_to_idx):
    """Create a pixel-level segmentation mask.
    Args:
        anns: list of LVIS annotation dicts
            (must include 'segmentation' and
             'category_id' fields)
        img_w, img_h: original image dimensions
        target_size: output mask size (e.g. 256)
        lvis_id_to_idx: dict mapping LVIS
            category IDs to class indices (0,1,2)
            The code adds +1 so mask labels are
            1,2,3 (0 is reserved for background)
    Returns:
        mask: np.ndarray (target_size,target_size)
            with integer class labels 0..K
    """
    # Start with all-background mask
    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
        # Class index: 1, 2, or 3
        # (0 = background)
        cls_idx = lvis_id_to_idx[cat_id] + 1

        seg = ann.get('segmentation', [])
        if not isinstance(seg, list):
            continue  # skip RLE-encoded masks
        for polygon in seg:
            if len(polygon) < 6:
                continue  # need >= 3 points
            # Convert flat list to (x,y) tuples
            poly_xy = list(zip(
                polygon[0::2], polygon[1::2]))
            draw.polygon(poly_xy,
                         fill=cls_idx)

    # Resize mask to target size using
    # NEAREST (do NOT interpolate class labels)
    mask = mask.resize(
        (target_size, target_size),
```

```
        Image.NEAREST)
    return np.array(mask)
```

Critical: When resizing segmentation masks, you **must** use nearest-neighbor interpolation (`Image.NEAREST`). Bilinear or bicubic interpolation would blend class labels at boundaries, creating invalid fractional labels.

Important notes:

- The mask uses class indices **1, 2, 3** for your foreground categories and **0 for background**. This means your segmentation network outputs $K + 1 = 4$ channels (background + 3 foreground classes).
- If two objects overlap in the image, the later annotation in the list will overwrite the earlier one in the mask. This is a standard simplification for semantic segmentation (which does not distinguish individual instances).
- You should save the masks alongside the images when constructing your dataset. Store them as single-channel PNG files to preserve integer class labels exactly.

Verification: For at least 3 sample images, display the original image and the corresponding segmentation mask side-by-side, using a distinct color for each class. Include this visualization in your report for Task 5.

3 Programming Tasks (100 Points Total)

3.1 Task 1: Single-Instance Detection (10 points)

Objective: Understand how a single network can simultaneously perform classification and bounding-box regression by using two different loss functions.

3.1.1 Running LOADnet2 on PurdueShapes5 (5 points)

Run the DLStudio example script:

```
object_detection_and_localization.py
```

This script trains `LOADnet2` on the PurdueShapes5 dataset (32×32 images), using `nn.CrossEntropyLoss` for classification and `nn.MSELoss` for bounding-box regression. Refer to Slides 50–73 of the multi-instance detection lecture [2].

Deliverables:

- Training loss curves for both the classification loss and the regression loss (two separate curves or overlaid with a legend)
- A grid of at least 6 test images showing: the original image, the ground-truth bounding box, and the predicted bounding box with class label
- Classification accuracy on the test set

3.1.2 Understanding Dual-Loss Training (5 points)

Report (answer in 1–2 paragraphs each):

- Explain the architecture of `LOADnet2`: how does the network branch into two output paths (classification head and regression head)? Where do the two paths diverge?
- Explain how two loss functions are backpropagated through the same network. Why must you be careful about when `backward()` is called? What does the `retain_graph=True` argument do, and why is it needed here? (See Slides 11–18 of the lecture [2].)
- Why is `nn.MSELoss` appropriate for bounding-box regression but not for classification? Why is `nn.CrossEntropyLoss` appropriate for classification but not for regression? (See Slides 38–49.)

Grading:

- Successful execution with all visual deliverables (5 points)
- Clear explanation of dual-loss training mechanics (5 points)

3.2 Task 2: Understanding YOLO Logic (10 points)

Objective: Understand the core concepts behind the YOLO approach to multi-instance object detection by studying the YOLOLogic module and running the multi-instance detection example.

3.2.1 Studying the YOLO Concepts (5 points)

Read Slides 103–118 of the multi-instance detection lecture [2] and study the YOLOLogic module source code. **Report** (answer each clearly):

- **Image cells:** Why is the image divided into a grid of cells? How does the grid structure help assign responsibility for detecting each object? What determines which cell is responsible for a given object?

- **Anchor boxes:** Why are multiple anchor boxes associated with each cell? What do different anchor boxes represent (hint: aspect ratios)? How is an anchor box assigned to a specific ground-truth object?

- **yolo_vector:** Describe the structure of a `yolo_vector`. What are its components (objectness, class probabilities, bounding-box offsets $\delta_x, \delta_y, \sigma_w, \sigma_h$)? Explain what each bounding-box parameter represents:

$$\delta_x = \frac{cx_{\text{GT}} - cx_{\text{cell}}}{s_{\text{cell}}}, \quad \delta_y = \frac{cy_{\text{GT}} - cy_{\text{cell}}}{s_{\text{cell}}} \tag{3}$$

$$w_{\text{GT}} = \sigma_w \cdot w_{\text{anchor}}, \quad h_{\text{GT}} = \sigma_h \cdot h_{\text{anchor}} \tag{4}$$

  where $(cx_{\text{GT}}, cy_{\text{GT}})$ is the center of the ground-truth box, $(cx_{\text{cell}}, cy_{\text{cell}})$ is the center of the assigned cell, and s_{cell} is the cell size in pixels.

- **yolo_tensor:** Explain the shape of the `yolo_tensor` and how it is organized. If you have C cells, A anchor boxes per cell, and each yolo_vector has length V, what is the shape of the yolo_tensor?

3.2.2 Running Multi-Instance Detection (5 points)

Run the YOLOLogic example script:

```
multi_instance_object_detection.py
```

This uses the `PurdueDrEvalMultiDataset` where each image contains multiple instances of "Dr. Eval", "house", and "watertower" with structured clutter and noise.

Deliverables:

- Training loss curves for all three losses: objectness (BCE), classification (CE), and bounding-box regression (MSE)

- A grid of at least 6 test images showing predicted bounding boxes and class labels alongside ground-truth annotations

- A brief paragraph discussing the quality of the multi-instance detections: Which objects are detected reliably? Where does the detector struggle?

Grading:

- Clear YOLO concept explanations (5 points)
- Successful multi-instance detection execution with deliverables (5 points)

3.3 Task 3: LVIS Dataset Construction (20 points)

Objective: Build both a **detection** dataset (with bounding boxes) and a **segmentation** dataset (with pixel-level masks) from LVIS for your 3 chosen categories. You will also implement IoU from scratch, which you will need in Tasks 4 and 5.

Important: Follow the requirements in Section 2.6 exactly. Both datasets must use the **same images and the same train/val split**.

3.3.1 Detection Dataset Construction (7 points)

Write a script that:

1. Loads LVIS annotations and filters images by your 3 chosen categories
2. Filters out objects with area $< 2{,}500$ pixels
3. Discards images with no qualifying foreground objects
4. Resizes images to 256×256 and scales bounding boxes accordingly
5. Splits into training (80%) and validation (20%) sets
6. Saves images and annotations in an organized directory structure

Annotation storage: For each image, store a JSON or text file with the following per-object entries: category index (0, 1, or 2), and bounding box `[x, y, w, h]` in the resized coordinate space.

Report:

- Include your complete dataset construction code
- Include the dataset verification output from Section 2.6
- Include the 3×3 sample image grid with drawn bounding boxes and labels

3.3.2 Segmentation Mask Construction (8 points)

Using the **same images and train/val split**, convert the LVIS polygon annotations into pixel-level segmentation masks. Follow the instructions in Section 2.7.

Requirements:

- Each mask should be a single-channel image of size 256×256 with pixel values in $\{0, 1, 2, 3\}$ (0 = background, 1–3 = your categories).
- Use nearest-neighbor interpolation when resizing masks.
- Save masks as single-channel PNG files alongside the images.

Report:

- Include your mask construction code
- Display at least **6 sample images** with their corresponding segmentation masks overlaid or shown side-by-side (2 per category), using distinct colors for each class
- Report the per-class pixel distribution: what fraction of all pixels in the training set belong to each class (including background)? *Note: background will likely dominate.*

3.3.3 IoU Computation (5 points)

Implement an Intersection over Union (IoU) function from scratch (**no library calls**). IoU measures the overlap between two bounding boxes:

$$\text{IoU}(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

where A and B are bounding boxes in `[x, y, w, h]` format.

Function signature:

```
def compute_iou(box1, box2):
    """Compute IoU between two boxes.
    Args:
        box1: [x, y, w, h] (top-left corner)
        box2: [x, y, w, h] (top-left corner)
    Returns:
        float: IoU value in [0, 1]
    """
```

Also implement a batched version:

```python
def compute_iou_matrix(boxes1, boxes2):
    """Compute pairwise IoU matrix.
    Args:
        boxes1: tensor (N, 4)
        boxes2: tensor (M, 4)
    Returns:
        tensor (N, M): pairwise IoU values
    """
```

You will use this IoU function in Task 4 for anchor assignment and evaluation. The autograder will test your implementation with known inputs and expected outputs.

Report:

- Include your complete IoU implementation code
- Show test cases demonstrating: (a) two non-overlapping boxes (IoU = 0), (b) two identical boxes (IoU = 1), (c) two partially overlapping boxes (IoU between 0 and 1)

Grading:

- Detection dataset construction with verification output and sample grid (7 points)
- Segmentation mask construction with visualizations and pixel distribution (8 points)
- Correct IoU implementation with test cases (5 points)

3.4 Task 4: YOLO Network for LVIS (30 points)

Objective: Implement a YOLO-style multi-instance object detection network for your LVIS dataset, train it with three loss functions, and evaluate it qualitatively.

3.4.1 Designing Your YOLO Parameters (5 points)

Before building the network, you must decide on the following YOLO parameters for your LVIS dataset:

- **Grid size:** How many cells to divide the 256×256 image into. A reasonable choice is a 4×4 or 8×8 grid (i.e., 16 or 64 cells).

- **Number of anchor boxes per cell (A):** How many anchor boxes to associate with each cell. Use $A = 5$ with aspect ratios `[1/5, 1/3, 1/1, 3/1, 5/1]` (matching the YOLOLogic module).
- **yolo_vector length (V):** Each yolo_vector contains: 1 objectness score + K class probabilities + 4 bounding-box offsets $= 1 + K + 4$, where $K = 3$ is your number of categories. So $V = 8$.
- **yolo_tensor shape:** For C total cells and A anchor boxes, the output tensor has shape (B, C, A, V) or equivalently $(B, C \cdot A \cdot V)$ when flattened.

Report:

- State your chosen grid size, number of anchor boxes, and the resulting yolo_tensor shape
- Show a diagram or clear description of how the yolo_tensor is organized

3.4.2 Network Architecture: LVISYOLONet (10 points)

Build a network called `LVISYOLONet` for multi-instance detection on your LVIS dataset. You may base your architecture on `NetForYOLO` from the YOLOLogic module, adapting it for 256×256 input images and your YOLO parameters.

Requirements:

- Input: $(B, 3, 256, 256)$ RGB images
- Output: $(B, C \cdot A \cdot V)$ flattened yolo_tensor, where C = number of cells, A = anchor boxes per cell, V = yolo_vector length
- Use skip connections (SkipBlock or similar) for the convolutional backbone
- The backbone should produce a feature map that is then mapped to the yolo_tensor through one or more fully connected or 1×1 convolutional layers

Architecture template:

```
class LVISYOLONet(nn.Module):
    def __init__(self, num_classes=3,
                 num_cells=16, num_anchors=5):
        super().__init__()
```

```
        self.num_classes = num_classes
        self.num_anchors = num_anchors
        self.num_cells = num_cells
        # yolo_vector: 1 + num_classes + 4
        self.yolo_vec_len = 1 + num_classes + 4
        self.out_size = (num_cells * num_anchors
                         * self.yolo_vec_len)

        # === CONVOLUTIONAL BACKBONE ===
        # (Use SkipBlocks for depth)

        # === OUTPUT HEAD ===
        # Map features to yolo_tensor
        self.fc = nn.Linear(???, self.out_size)

    def forward(self, x):
        # === BACKBONE FORWARD ===
        x = self.fc(x)
        return x
```

Report:

- Include your complete `LVISYOLONet` code
- State the total number of learnable parameters

3.4.3 Custom Dataloader (10 points)

Write a custom `Dataset` class that returns the following from `__getitem__`:

1. The image tensor (after transforms)
2. For each foreground object in the image:
   (a) The index of the assigned cell (the cell whose center is closest to the object center)
   (b) The index of the assigned anchor box (the anchor with the highest IoU with the ground-truth bounding box)
   (c) The ground-truth yolo_vector (objectness=1, one-hot class label, and bounding-box offsets $\delta_x, \delta_y, \sigma_w, \sigma_h$)

Use your IoU function from Section 3.3.3 for anchor box assignment.
Cell and anchor assignment template:

```python
def assign_cell_and_anchor(bbox, img_size,
        grid_size, anchor_ratios):
    """Assign a GT box to a cell and anchor.
    Args:
        bbox: [x, y, w, h] in pixel coords
        img_size: image dimension (256)
        grid_size: e.g. 4 for a 4x4 grid
        anchor_ratios: list of h/w ratios,
            e.g. [1/5, 1/3, 1, 3, 5]
    Returns:
        cell_idx: int (flattened cell index)
        anchor_idx: int
        offsets: [delta_x, delta_y,
                  sigma_w, sigma_h]
    """
    x, y, w, h = bbox
    # Object center
    cx = x + w / 2.0
    cy = y + h / 2.0

    # Cell size in pixels
    cell_size = img_size / grid_size

    # Find which cell the center falls in
    col = int(cx / cell_size)
    row = int(cy / cell_size)
    col = min(col, grid_size - 1)
    row = min(row, grid_size - 1)
    cell_idx = row * grid_size + col

    # Cell center in pixels
    cell_cx = (col + 0.5) * cell_size
    cell_cy = (row + 0.5) * cell_size

    # Offsets: distance from cell center
    delta_x = (cx - cell_cx) / cell_size
    delta_y = (cy - cell_cy) / cell_size

    # Find best anchor by IoU
    # Anchor boxes are centered at cell center
    # with different aspect ratios
    best_anchor = 0
    best_iou = -1
    anchor_area = cell_size * cell_size
    for a_idx, ratio in enumerate(anchor_ratios):
        # anchor dims from aspect ratio
        a_h = (anchor_area * ratio) ** 0.5
        a_w = a_h / ratio
        # anchor box centered at cell center
```

```
        a_box = [cell_cx - a_w/2,
                 cell_cy - a_h/2, a_w, a_h]
        iou = compute_iou(bbox, a_box)
        if iou > best_iou:
            best_iou = iou
            best_anchor = a_idx
            best_a_w, best_a_h = a_w, a_h

    # Size ratios
    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]
```

Important: The cell/anchor assignment logic is one of the most error-prone parts of this homework. Test it thoroughly by visualizing the assigned cells and anchors for several ground-truth boxes before training.

Report: Include your complete dataloader code with comments explaining the cell and anchor assignment logic.

Ground-truth yolo_tensor construction template:

The following shows how to build the target tensor for one image. This is part of your `__getitem__` method.

```
def build_gt_yolo_tensor(anns, img_size,
        grid_size, anchor_ratios, num_classes):
    """Build ground-truth yolo_tensor for
    one image.
    Args:
        anns: list of {category_idx, bbox}
        img_size, grid_size, anchor_ratios:
            YOLO params
        num_classes: number of categories
    Returns:
        gt_tensor: (num_cells, num_anchors,
                    1 + num_classes + 4)
    """
    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)

    for ann in anns:
        cls_idx = ann['category_idx']
        bbox = ann['bbox']
```

```
        cell_idx, anchor_idx, offsets = \
            assign_cell_and_anchor(
                bbox, img_size, grid_size,
                anchor_ratios)

        # objectness = 1
        gt[cell_idx, anchor_idx, 0] = 1.0
        # one-hot class label
        gt[cell_idx, anchor_idx,
           1 + cls_idx] = 1.0
        # bounding box offsets
        gt[cell_idx, anchor_idx,
           1+num_classes:] = \
            torch.tensor(offsets)

    return gt
```

Note: If two objects map to the same cell and anchor, the second one will overwrite the first. This is a known limitation of the basic YOLO approach. For this homework, this is acceptable.

3.4.4 Training and Evaluation (5 points)

Train `LVISYOLONet` using three loss functions:

- **Objectness loss:** `nn.BCEWithLogitsLoss`, binary classification of whether each cell-anchor combination contains an object
- **Classification loss:** `nn.CrossEntropyLoss`, which category the detected object belongs to (computed only for cell-anchor combinations that contain an object)
- **Bounding box regression loss:** `nn.MSELoss`, offset between predicted and ground-truth bounding box parameters (computed only for cell-anchor combinations that contain an object)

Training configuration:

- Optimizer: `Adam(lr=1e-4)`
- Epochs: 15–20
- Batch size: 16
- Image size: 256×256

Deliverables:

- Plot all three loss curves over training iterations (overlaid on one figure with a legend)
- Display multi-instance detection results for at least **12 images** from the **validation set**:
  - For each image, show the predicted bounding boxes and class labels alongside the ground-truth annotations
  - Include at least **8 good detections** where the detector works well (try to cover all 3 categories)
  - Include at least **4 failure cases** illustrating current shortcomings (missed objects, wrong labels, poor localization)
- A paragraph discussing the detector's performance: What works? What fails? Which categories are easier/harder to detect?

Decoding predictions into bounding boxes:

To visualize results, you must convert the flat network output back into bounding boxes. The following template outlines the steps:

```
def decode_yolo_output(output, img_size,
        grid_size, anchor_ratios,
        num_classes=3, conf_thresh=0.5):
    """Convert network output to boxes.
    Args:
        output: tensor (num_cells*A*V,)
        img_size, grid_size, anchor_ratios:
            YOLO params
        num_classes: number of categories
        conf_thresh: objectness threshold
    Returns:
        boxes: list of [x,y,w,h]
        labels: list of int class indices
        scores: list of float confidences
    """
    num_anchors = len(anchor_ratios)
    num_cells = grid_size * grid_size
    vec_len = 1 + num_classes + 4
    pred = output.view(
        num_cells, num_anchors, vec_len)

    cell_size = img_size / grid_size
    boxes, labels, scores = [], [], []

    for c_idx in range(num_cells):
        row = c_idx // grid_size
```

```
        col = c_idx % grid_size
        cell_cx = (col + 0.5) * cell_size
        cell_cy = (row + 0.5) * cell_size

        for a_idx in range(num_anchors):
            vec = pred[c_idx, a_idx]
            obj_score = torch.sigmoid(vec[0])
            if obj_score < conf_thresh:
                continue

            # Class prediction
            cls_probs = vec[1:1+num_classes]
            cls_idx = cls_probs.argmax().item()

            # Decode bounding box
            dx = vec[1+num_classes]
            dy = vec[2+num_classes]
            sw = vec[3+num_classes]
            sh = vec[4+num_classes]
            cx = cell_cx + dx * cell_size
            cy = cell_cy + dy * cell_size
            # Reconstruct anchor dims
            ratio = anchor_ratios[a_idx]
            area = cell_size * cell_size
            a_h = (area * ratio) ** 0.5
            a_w = a_h / ratio
            w = sw * a_w
            h = sh * a_h
            x = cx - w / 2
            y = cy - h / 2

            boxes.append([x, y, w, h])
            labels.append(cls_idx)
            scores.append(obj_score.item())

    return boxes, labels, scores
```

Grading:

- YOLO parameter design and clear explanation (5 points)
- Correct `LVISYOLONet` architecture (10 points)
- Complete dataloader with cell/anchor assignment (10 points)
- Training with loss curves and qualitative evaluation (12 images) (5 points)

3.5 Task 5: Semantic Segmentation on LVIS (30 points)

Objective: Understand encoder-decoder architectures and transpose convolutions, first through DLStudio's mUnet on a simple dataset, then by training a segmentation network on your LVIS data, the same 3 categories and images you used for object detection in Task 4.

3.5.1 Understanding Transpose Convolutions and mUnet Warm-up (5 points)

Transpose convolutions are the key operation that enables the decoder to upsample feature maps back to the original image resolution. Study Slides 29–67 of the semantic segmentation lecture [3], and study the `SemanticSegmentation` inner class in `DLStudio/DLStudio.py`, specifically the `mUnet` network and its building blocks (`SkipBlockDN` for the encoder, `SkipBlockUP` for the decoder).

Report (answer each):

- Explain how a standard 2D convolution can be expressed as a matrix-vector product. How does this representation lead to the definition of a transpose convolution?
- Given an input of size 4×4 and a transpose convolution with kernel size 3, stride 2, and padding 1, what is the output size? Show your calculation using the formula:

$$H_{\text{out}} = (H_{\text{in}} - 1) \times \text{stride} - 2 \times \text{padding} + \text{kernel\_size}$$

- What is the difference between `nn.ConvTranspose2d` and simple upsampling followed by a regular convolution (e.g., `nn.Upsample` + `nn.Conv2d`)? Which approach has learnable parameters?
- Draw or describe the overall encoder-decoder architecture of mUnet: how many downsampling stages, how many upsampling stages, where are the skip connections? How do these encoder-decoder skip connections differ in purpose from the ResNet skip connections you studied in HW5?

Run the DLStudio semantic segmentation example on PurdueShapes5MultiObject to verify your understanding:

```
semantic_segmentation.py
```

Deliverables:

- Training loss curve for mUnet on PurdueShapes5MultiObject
- For at least **4 test images**, display side-by-side: the original input, the ground-truth mask, and the predicted mask
- A brief paragraph noting whether mUnet produces reasonable segmentations on this simple dataset

3.5.2 Training LVISSegNet on LVIS (10 points)

Build and train a network called `LVISSegNet` for semantic segmentation on your LVIS dataset, using the segmentation masks you constructed in Task 3 (Section 3.3.2). You may base your architecture on DLStudio's `mUnet`, adapting it for 256×256 images and $K + 1 = 4$ output classes.

Requirements:

- Input: $(B, 3, 256, 256)$ RGB images
- Output: $(B, 4, 256, 256)$, per-pixel logits for 4 classes (background + 3 categories)
- Must use an encoder-decoder structure with transpose convolutions for upsampling
- Must use skip connections between encoder and decoder

Architecture starter code:

The following provides a largely complete encoder-decoder architecture based on DLStudio's mUnet, adapted for 256×256 input. You must **fill in the marked sections** and may modify channel counts, number of stages, or other hyperparameters. **Cite DLStudio** wherever you reuse code.

```
import torch
import torch.nn as nn

class SkipBlockDN(nn.Module):
    """Encoder block: conv -> BN -> ReLU
       with skip connection, then downsample."""
    def __init__(self, in_ch, out_ch):
        super().__init__()
        self.conv1 = nn.Conv2d(in_ch, out_ch,
                     3, padding=1)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.conv2 = nn.Conv2d(out_ch, out_ch,
```

```
                      3, padding=1)
        self.bn2 = nn.BatchNorm2d(out_ch)
        self.pool = nn.MaxPool2d(2, 2)
        # 1x1 conv for skip if channels change
        self.skip = nn.Conv2d(in_ch, out_ch, 1) \
                    if in_ch != out_ch else \
                    nn.Identity()

    def forward(self, x):
        skip = self.skip(x)
        x = torch.relu(self.bn1(self.conv1(x)))
        x = self.bn2(self.conv2(x))
        x = torch.relu(x + skip)
        return self.pool(x), x  # pooled, skip

class SkipBlockUP(nn.Module):
    """Decoder block: transpose conv to
       upsample, then concat skip, then conv."""
    def __init__(self, in_ch, out_ch):
        super().__init__()
        self.up = nn.ConvTranspose2d(
            in_ch, out_ch, 2, stride=2)
        # After concat: out_ch (from up) +
        #               in_ch (from skip)
        self.conv1 = nn.Conv2d(out_ch + in_ch,
                    out_ch, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.conv2 = nn.Conv2d(out_ch, out_ch,
                    3, padding=1)
        self.bn2 = nn.BatchNorm2d(out_ch)

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

class LVISSegNet(nn.Module):
    def __init__(self, num_classes=4):
        super().__init__()
        # Encoder: 256->128->64->32->16
        self.enc1 = SkipBlockDN(3, 64)
        self.enc2 = SkipBlockDN(64, 128)
        self.enc3 = SkipBlockDN(128, 256)
        self.enc4 = SkipBlockDN(256, 512)
        # Bottleneck at 16x16
        ##,-- FILL IN,--
        ## Add a bottleneck conv layer(s) at
```

```
        ## the lowest resolution (16x16)
        ##,------------------

        # Decoder: 16->32->64->128->256
        self.dec4 = SkipBlockUP(512, 256)
        self.dec3 = SkipBlockUP(256, 128)
        self.dec2 = SkipBlockUP(128, 64)
        self.dec1 = SkipBlockUP(64, 64)
        # Final 1x1 conv
        self.final = nn.Conv2d(
            64, num_classes, 1)

    def forward(self, x):
        # Encoder (save skip connections)
        x, s1 = self.enc1(x)   # 128x128
        x, s2 = self.enc2(x)   # 64x64
        x, s3 = self.enc3(x)   # 32x32
        x, s4 = self.enc4(x)   # 16x16
        ##,-- FILL IN,--
        ## Pass x through bottleneck
        ##,------------------

        # Decoder (use skip connections)
        x = self.dec4(x, s4)   # 32x32
        x = self.dec3(x, s3)   # 64x64
        x = self.dec2(x, s2)   # 128x128
        x = self.dec1(x, s1)   # 256x256
        return self.final(x)

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

What you must do:

- Fill in the bottleneck layer(s) at the lowest resolution. This can be as simple as one or two `Conv2d + BN + ReLU` layers at 16×16.
- Verify that the network produces the correct output shape by running: `LVISSegNet()(torch.randn(1,3,256,256)).shape` $\rightarrow$ `(1, 4, 256, 256)`
- You are free to modify channel counts, add dropout, change the number of encoder/decoder stages, etc. The starter code is a working baseline, not a constraint.

Training configuration:

- Loss: `nn.CrossEntropyLoss` applied per-pixel. The target mask should have shape $(B, 256, 256)$ with integer class labels. **Recommended:** Use `weight` parameter to handle class imbalance (background pixels will heavily outnumber foreground pixels).
- Optimizer: `Adam(lr=1e-3)`
- Epochs: 10–15
- Batch size: 8–16
- Image size: 256×256

Class weighting example:

```
# Compute inverse-frequency weights
class_counts = [bg_pixels, c1_pixels,
                c2_pixels, c3_pixels]
total = sum(class_counts)
weights = [total / (4 * c)
           for c in class_counts]
weights = torch.tensor(weights,
                       dtype=torch.float)
criterion = nn.CrossEntropyLoss(weight=weights)
```

Deliverables:

- Include your complete `LVISSegNet` code
- Total number of learnable parameters
- Training and validation loss curves
- For at least **8 validation images**, display side-by-side:
  1. The original input image
  2. The ground-truth segmentation mask
  3. The predicted segmentation mask
- Per-class pixel accuracy and overall pixel accuracy:

$$\text{Pixel Acc}_c = \frac{\#\text{ pixels correctly classified as } c}{\#\text{ pixels with GT label } c}$$

- Mean IoU (mIoU) across the 3 foreground classes:

$$\text{IoU}_c = \frac{\text{TP}_c}{\text{TP}_c + \text{FP}_c + \text{FN}_c}$$

where TP_c = pixels correctly labeled c, FP_c = pixels wrongly labeled c, FN_c = pixels of class c labeled as something else.

- A paragraph discussing: How well does the network segment each category? Are boundaries crisp or blurry? Does background domination affect foreground segmentation quality?

3.5.3 Quantitative Evaluation and Master Summary (15 points)

YOLO Precision/Recall Evaluation (7 points):

Using your IoU function from Task 3, implement a quantitative evaluation of your YOLO detector on the validation set:

1. For each validation image, convert the predicted yolo_tensor into bounding boxes using the decode function. Apply a confidence threshold (e.g., 0.5).
2. A prediction is a **true positive** if its IoU with a ground-truth box of the same class exceeds 0.5; otherwise it is a **false positive**. Unmatched ground-truth boxes are **false negatives**.
3. Compute and report **per-class precision and recall**:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{5}$$

Report:

- Include your evaluation code
- Report precision and recall per class in a table

Master Summary Table and Discussion (8 points):

Create a table covering all your experiments:

Network	Dataset	Parameters	Key Metric
LOADnet2	PurdueShapes5	?	Classif. accuracy: ?
NetForYOLO	PurdueDrEvalMulti	?	Qualitative
LVISYOLONet	LVIS (3 cat)	?	Per-class P/R: ?
mUnet	PurdueShapes5Multi	?	Pixel accuracy: ?
LVISSegNet	LVIS (3 cat)	?	mIoU: ?

Write a 2–3 paragraph summary of your key takeaways. Address:

- What is the most challenging aspect of multi-instance detection compared to classification?
- How do the three YOLO losses interact during training? Did any of them dominate?
- What did you learn about encoder-decoder architectures and pixel-level prediction?
- Having applied both detection and segmentation to the *same* LVIS data, briefly compare: what does each approach capture that the other misses?

Grading:

- Transpose convolution explanations and mUnet warm-up (5 points)
- LVISSegNet training with all deliverables including mIoU (10 points)
- YOLO P/R evaluation, master summary table, and overall discussion (15 points)

4 Bonus Task (20 Points)

4.1 DIoU Loss for Bounding Box Regression

Replace the MSE loss for bounding box regression in your `LVISYOLONet` (Task 4) with the **Distance-IoU (DIoU) loss** [8]. The DIoU loss is defined as:

$$\mathcal{L}_{\text{DIoU}} = 1 - \text{IoU} + \frac{\rho^2(\mathbf{b}, \mathbf{b}^{\text{gt}})}{c^2}$$

where $\rho(\mathbf{b}, \mathbf{b}^{\text{gt}})$ is the Euclidean distance between the centers of the predicted and ground-truth boxes, and c is the diagonal length of the smallest enclosing rectangle of both boxes.

You may reference DLStudio's `object_detection_and_localization_iou.py` example as a starting point.

Deliverables:

1. Your DIoU loss implementation code
2. Training loss curves for the DIoU variant (same format as Task 4)
3. At least 12 annotated validation images (same format as Task 4)

4. A comparison table: MSE-based regression vs. DIoU-based regression, showing per-class precision and recall at IoU threshold 0.5

5. A paragraph discussing whether DIoU loss improved bounding box localization quality

5 Submission Instructions

5.1 What to Submit in the Report

Read this section carefully. Incomplete submissions will lose points. Every item listed below must be present in your submission.

1. **Task 1: Single-Instance Detection Warm-up (10 points)**
   (a) **Code:** The script you used to run LOADnet2
   (b) **Figure:** Training loss curves (classification and regression)
   (c) **Figure:** Grid of test images with predicted and ground-truth bounding boxes
   (d) **Text:** Classification accuracy
   (e) **Text:** Explanation of dual-loss training: network branching, `retain_graph`, and why MSE vs. CE for different tasks

2. **Task 2: Understanding YOLO Logic (10 points)**
   (a) **Text:** Explanation of image cells, anchor boxes, yolo_vector, and yolo_tensor
   (b) **Code:** Script used to run multi-instance detection example
   (c) **Figure:** Three training loss curves (objectness, classification, regression)
   (d) **Figure:** Grid of test images with multi-instance predictions and ground truth
   (e) **Text:** Discussion of detection quality

3. **Task 3: LVIS Dataset Construction (20 points)**
   (a) **Code:** Complete detection dataset construction script
   (b) **Output:** Dataset verification (image counts, instance counts per category, average objects per image, category names)

(c) **Figure:** 3×3 sample image grid with bounding boxes and labels

(d) **Code:** Segmentation mask construction code

(e) **Figure:** At least 6 sample images with segmentation masks (2 per category)

(f) **Text:** Per-class pixel distribution in training set

(g) **Code:** IoU implementation (`compute_iou` and `compute_iou_matrix`)

(h) **Output:** IoU test cases (non-overlapping, identical, partial overlap)

4. **Task 4: YOLO Network for LVIS (30 points)**

(a) **Text:** YOLO parameter choices (grid size, anchors, yolo_tensor shape) with explanation

(b) **Code:** Complete `LVISYOLONet` class definition

(c) **Text:** Total learnable parameters

(d) **Code:** Custom dataloader with cell/anchor assignment logic

(e) **Code:** Training script showing the three-loss training loop

(f) **Figure:** All three loss curves overlaid on one plot

(g) **Figure:** At least 12 annotated validation images (8 good + 4 failure cases)

(h) **Text:** Performance discussion paragraph

5. **Task 5: Semantic Segmentation on LVIS (30 points)**

(a) **Text:** Transpose convolution explanations (matrix formulation, output size calculation, comparison with upsampling, encoder-decoder skip connections vs. ResNet skip connections)

(b) **Code:** Script used to run mUnet on PurdueShapes5MultiObject

(c) **Figure:** mUnet training loss curve and at least 4 test images (input, GT mask, predicted mask)

(d) **Code:** Complete `LVISSegNet` class definition (with filled-in bottleneck)

(e) **Text:** Total learnable parameters for `LVISSegNet`

(f) **Figure:** Training and validation loss curves for `LVISSegNet`

(g) **Figure:** At least 8 validation images with input, GT mask, and predicted mask

(h) **Table:** Per-class pixel accuracy, overall pixel accuracy, and mIoU

(i) **Text:** Discussion of segmentation quality

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

(k) **Table:** Per-class precision and recall at IoU ≥ 0.5

(l) **Table:** Master summary table (5 rows: LOADnet2, NetForYOLO, LVISYOLONet, mUnet, LVISSegNet)

(m) **Text:** 2–3 paragraph overall discussion and takeaways

6. **Bonus: DIoU Loss (20 points, optional)**

   (a) **Code:** DIoU loss implementation

   (b) **Figure:** Training loss curves for DIoU variant

   (c) **Figure:** 12 annotated validation images

   (d) **Table:** MSE vs. DIoU comparison (precision/recall per class)

   (e) **Text:** Discussion of DIoU vs. MSE for bounding box regression

5.2 Code Files to Submit

Submit the following (see Section 5.3 for exact function signatures):

1. `models.py`: `LVISYOLONet`, `LVISSegNet`
2. `dataset.py`: LVIS detection dataset class, segmentation mask construction, and dataset building script
3. `train_yolo.py`: YOLO training loop with three losses
4. `train_seg.py`: Segmentation training loop
5. `evaluate.py`: `compute_iou`, `compute_iou_matrix`, precision/recall evaluation, pixel accuracy and mIoU functions
6. `visualize.py`: Functions for drawing bounding boxes and segmentation masks
7. `requirements.txt`: Dependencies
8. `README.md`: Instructions to run your code

5.3 Autograder Interface Requirements

The autograder will import your files and call specific functions. If your file names or function signatures do not match exactly, the autograder will fail and you will lose points.

5.3.1 Required File Names

Submit exactly these file names (case-sensitive):

- `models.py`
- `evaluate.py`
- `dataset.py`

5.3.2 `models.py`

Must contain:

- `LVISYOLONet(num_classes=3, num_cells=16, num_anchors=5)`: YOLO-style network. `forward(x)` accepts $(B, 3, 256, 256)$ and returns $(B, C \cdot A \cdot V)$. Must have `count_parameters()` returning an `int`.
- `LVISSegNet(num_classes=4)`: Encoder-decoder segmentation network. `forward(x)` accepts $(B, 3, 256, 256)$ and returns $(B, 4, 256, 256)$, per-pixel logits for 4 classes. Must have `count_parameters()` returning an `int`.

5.3.3 `evaluate.py`

Must contain:

- `compute_iou(box1, box2)`: Takes two lists/tensors of 4 elements `[x, y, w, h]`, returns a float.
- `compute_iou_matrix(boxes1, boxes2)`: Takes tensors of shape $(N, 4)$ and $(M, 4)$, returns tensor of shape (N, M).

The autograder will run:

```
from models import LVISYOLONet, LVISSegNet
from evaluate import compute_iou, \
                     compute_iou_matrix
import torch
```

```
5
6  # Test LVISYOLONet
7  net = LVISYOLONet(num_classes=3,
8                    num_cells=16,
9                    num_anchors=5)
10 out = net(torch.randn(2, 3, 256, 256))
11 vec_len = 1 + 3 + 4  # 8
12 assert out.shape == (2, 16 * 5 * vec_len)
13
14 # Test LVISSegNet
15 seg = LVISSegNet(num_classes=4)
16 out_seg = seg(torch.randn(2, 3, 256, 256))
17 assert out_seg.shape == (2, 4, 256, 256)
18
19 # Test IoU: non-overlapping
20 iou = compute_iou([0,0,10,10], [20,20,10,10])
21 assert abs(iou - 0.0) < 1e-6
22
23 # Test IoU: identical
24 iou = compute_iou([5,5,10,10], [5,5,10,10])
25 assert abs(iou - 1.0) < 1e-6
26
27 # Test IoU: partial overlap
28 iou = compute_iou([0,0,10,10], [5,5,10,10])
29 expected = 25.0 / 175.0  # 0.142857...
30 assert abs(iou - expected) < 1e-4
31
32 # Test IoU matrix
33 b1 = torch.tensor([[0,0,10,10],
34                    [5,5,10,10]],
35                   dtype=torch.float)
36 b2 = torch.tensor([[0,0,10,10],
37                    [20,20,10,10]],
38                   dtype=torch.float)
39 mat = compute_iou_matrix(b1, b2)
40 assert mat.shape == (2, 2)
41 assert abs(mat[0,0].item() - 1.0) < 1e-6
42 assert abs(mat[0,1].item() - 0.0) < 1e-6
```

5.3.4 Autograder Summary

File	Required Names	Autograder Tests
`models.py`	`LVISYOLONet`, `LVISSegNet`	Forward pass shapes, parameter count
`evaluate.py`	`compute_iou`, `compute_iou_matrix`	Known-input IoU tests, matrix shape check

Important Notes:

- Do **not** rename these files or functions.
- Do **not** put executable code at the module level. Guard scripts with `if __name__ == '__main__':` blocks.
- Default argument values must match those shown above.
- **Cite DLStudio and YOLOLogic source code** wherever you borrow or adapt code.

5.4 Code Quality Requirements

- Well-commented code explaining key steps
- Meaningful variable and function names
- Docstrings for all classes and functions
- Code should run without modification (given correct paths)
- Follow PEP 8 style guidelines

5.5 Report Formatting

- **Include code snippets in the report for every task.** For each task, show the code you wrote alongside the outputs it produced. If the code snippet is not present, you will lose points on that task.
- Use figures and tables with captions
- Number equations, figures, and tables
- Use proper citations when referencing papers or DLStudio/YOLOLogic

5.6 Additional Guidelines

- Convert Jupyter notebooks to .py files. **Do NOT submit .ipynb**
- If submitting late, submit only once
- **Do NOT submit dataset files**, only code
- Your code and report must be your own work

6 Frequently Asked Questions (FAQ)

Q: Can I reuse code from previous homeworks?

A: Yes. You are encouraged to reuse and extend your previous implementations (e.g., dataset loading utilities, plotting functions).

Q: Can I use a pretrained backbone (e.g., ResNet) for LVISY-OLONet?

A: No. Build your backbone from scratch using SkipBlocks or similar. The goal is to understand the architecture, not to achieve state-of-the-art performance.

Q: My YOLO detector is not detecting any objects.

A: (1) Check that your dataloader correctly assigns objects to cells and anchors. (2) Verify that the objectness labels are correct. (3) Start with a small dataset to overfit and verify the pipeline works. (4) Lower the confidence threshold when converting predictions to bounding boxes.

Q: How do I convert the yolo_tensor output into bounding box predictions?

A: (1) Reshape the flat output into (C, A, V). (2) Apply sigmoid to the objectness score. (3) For cell-anchor combinations with objectness above your threshold, extract the class label (argmax of class probabilities) and bounding box offsets. (4) Convert offsets back to absolute image coordinates using the cell position and anchor box dimensions.

Q: Which 3 LVIS categories should I choose?

A: Choose categories that have enough images with multiple instances and bounding box area $\geq$ 2,500 pixels. The suggested sets in Section 2.6 are good starting points. If a category has too few qualifying images, pick a different one and document the change.

Q: I get a `KeyError` when looking up my category name.

A: LVIS uses verbose, synset-style category names, for example, `'car_(automobile)'` instead of `'car'`. Browse `lvis_data['categories']` to find exact names. A quick way:

```
for cat in lvis_data['categories']:
    if 'car' in cat['name'].lower():
        print(cat['name'], cat['id'],
              cat['instance_count'])
```

Q: I get a `KeyError` for `img_info['file_name']`.

A: LVIS image entries do not have a `file_name` field. They have a `coco_url` field (e.g., `http://.../train2017/000000391895.jpg`). Extract the filename with `img_info['coco_url'].split('/')[-1]`.

Q: My images have very few multi-instance examples.

A: Some LVIS categories naturally have more multi-instance images than others. Try categories like `car_(automobile)`, `chair`, `book`, `bottle`, `cup` which frequently appear in groups.

Q: Should I use the same 5 LVIS categories from HW4/HW5?

A: Not necessarily. This homework requires only 3 categories, and they must have bounding box annotations with sufficient area. You may choose different categories from HW4/HW5. Document your choices clearly.

Q: The three YOLO losses are on very different scales. Should I weight them?

A: Yes, it is common to weight the losses. A reasonable starting point is equal weights, but you may need to adjust. For example, if the regression loss dominates, reduce its weight. Document any loss weighting you use.

Q: My mUnet training is very slow.

A: mUnet on PurdueShapes5MultiObject should be manageable. Use a GPU if available. Reduce batch size if GPU memory is insufficient. Use Google Colab's free GPU if needed.

Q: What is the PurdueShapes5MultiObject dataset?

A: It is a variant of PurdueShapes5 where each 64×64 image can contain multiple shape objects, and the ground truth includes pixel-level segmentation masks. It is included in the DLStudio datasets download. See Slides 72–80 of the semantic segmentation lecture [3].

Q: My IoU function gives negative values.

A: Ensure you clamp the intersection dimensions to be non-negative. If two boxes do not overlap, the intersection area should be 0, not negative.

Q: Can I use `torchvision.ops.box_iou`?

A: **No.** You must implement IoU from scratch. The autograder will test your implementation.

Q: How do I compute per-class pixel accuracy for semantic segmentation?

A: For each class c, count the number of pixels where both the prediction and ground truth are c (true positives for that class), and divide by the total number of pixels where the ground truth is c.

Q: I don't have a GPU. Can I still do this homework?

A: The PurdueShapes5 and PurdueDrEval experiments are designed to run on CPU. For `LVISYOLONet`, training on CPU will be slow but feasible with reduced epochs. Use Google Colab's free GPU if needed.

Q: Can I work in a group?

A: No. This is individual work.

Q: How much of NetForYOLO can I reuse?

A: You may use it as a reference and adapt it for your LVIS dataset, but you must modify it for 256×256 inputs and your YOLO parameters. Cite the YOLOLogic source wherever you borrow code.

Q: My training loss for objectness is very high.

A: This is expected early in training because most cell-anchor combinations do not contain objects (the dataset is sparse). The objectness loss should decrease over epochs. If it stays very high, check your label construction in the dataloader.

Q: Do I need to implement Non-Maximum Suppression (NMS)?

A: NMS is not required but is recommended for cleaner visualizations. If multiple anchor boxes predict the same object, NMS removes duplicate detections. You may implement a simple version or use `torchvision.ops.nms` for visualization purposes only.

Q: Can I use the same LVISSegNet architecture as mUnet without changes?

A: You should adapt it for 256×256 input (mUnet uses 64×64) and for 4 output classes. You may keep the overall encoder-decoder structure similar, but you will likely need to adjust the number of layers or channels. Cite DLStudio wherever you borrow code.

Q: My segmentation masks have artifacts from polygon conversion.

A: This is expected for complex shapes. Ensure you are using `Image.NEAREST` for resizing masks. Small artifacts at boundaries are acceptable.

Q: My segmentation network predicts everything as background.

A: Background pixels heavily outnumber foreground pixels. Use class-weighted `CrossEntropyLoss` as described in the Task 5 template. You can also try focal loss if weighting alone is not sufficient.

Q: Some LVIS annotations have empty or invalid segmentation polygons.

A: Skip annotations with empty `segmentation` fields. Some annotations may have `segmentation` as an RLE (run-length encoding) dict instead of a polygon list; you may skip those or decode them using `pycocotools.mask.decode`.

Q: LVISSegNet is running out of GPU memory.

A: Encoder-decoder networks at 256×256 can be memory-hungry. (1) Reduce batch size to 4–8. (2) Reduce the number of channels. (3) Use fewer encoder/decoder stages. (4) Use Google Colab's free GPU.

Q: How do I compute mIoU?

A: For each foreground class c, compute $\text{IoU}_c = \text{TP}_c/(\text{TP}_c + \text{FP}_c + \text{FN}_c)$ where TP, FP, FN are pixel counts. Then $\text{mIoU} = \frac{1}{3}\sum_{c=1}^{3} \text{IoU}_c$ (average over the 3 foreground classes only, excluding background). .

References

[1] Avinash Kak, *DLStudio Platform*. Available at: https://engineering.purdue.edu/kak/distDLS/

[2] Avinash Kak, *Multi-Instance Object Detection, Image Cells and Anchor Boxes*. Available at: https://engineering.purdue.edu/kak/pdf-kak/MultiDetection.pdf

[3] Avinash Kak, *Transpose Convolutions and the Encoder-Decoder Architectures for Semantic Segmentation of Images*. Available at: https://engineering.purdue.edu/kak/pdf-kak/SemanticSeg.pdf

[4] Avinash Kak, *Using Skip Connections to Mitigate the Problem of Vanishing Gradients, and Using Batch, Instance, and Layer Normalizations for Improved SGD in Deep Networks*. Available at: https://engineering.purdue.edu/kak/pdf-kak/SkipConsAndBN.pdf

[5] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi, *You Only Look Once: Unified, Real-Time Object Detection*, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016. Available at: https://arxiv.org/abs/1506.02640

[6] Olaf Ronneberger, Philipp Fischer, and Thomas Brox, *U-Net: Convolutional Networks for Biomedical Image Segmentation*, MICCAI, 2015. Available at: https://arxiv.org/abs/1505.04597

[7] Agrim Gupta, Piotr Dollár, and Ross Girshick, *LVIS: A Dataset for Large Vocabulary Instance Segmentation*, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2019. Available at: https://arxiv.org/abs/1908.03195

[8] Zhaohui Zheng, Ping Wang, Wei Liu, Jinze Li, Rongguang Ye, and Dongwei Ren, *Distance-IoU Loss: Faster and Better Learning for Bounding Box Regression*, AAAI Conference on Artificial Intelligence, 2020. Available at: https://arxiv.org/abs/1911.08287

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, *Deep Residual Learning for Image Recognition*, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016. Available at: https://arxiv.org/abs/1512.03385