BME646 and ECE60146: Homework 4

Spring 2026
Due Date: Tuesday, Feb 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.**

1 Introduction

This homework has two main goals. First, you will develop a working understanding of `torch.nn` by exploring the DLStudio package and working with a CNN from scratch using the `torch.nn.Module` API. Second, you will familiarize yourself with the LVIS (Large Vocabulary Instance Segmentation) dataset and create three curated image subsets of varying complexity.

Concretely, you will:

- Install and explore DLStudio, running its built-in CIFAR-10 examples to understand how `torch.nn` is used to define, train, and evaluate CNNs
- Learn the LVIS API and use it to query categories, annotations, and images
- Create three classification datasets from LVIS with different properties (single-instance, multi-instance same category, multi-instance different categories)
- Implement a CNN using `torch.nn.Module`, train it on each dataset, and compute confusion matrices

1.1 Why LVIS

The LVIS (Large Vocabulary Instance Segmentation) dataset [2] provides several advantages for this course:

- **Rich vocabulary:** 1,203 object categories, enabling fine-grained classification studies

- **Federated annotations:** Categories organized by frequency (rare/-common/frequent) for balanced sampling
- **Instance-level annotations:** Precise instance counts and segmentation masks per image
- **Built on COCO images:** Uses COCO 2017 images with richer annotations
- **Standard API:** Compatible with `pycocotools`, a widely used toolset worth learning

LVIS provides significantly more categories than other common benchmarks, allowing you to work with fine-grained object distinctions. Learning to navigate its API and annotation structure is directly transferable to other large-scale vision datasets you will encounter in research and industry.

DLStudio Documentation: https://engineering.purdue.edu/kak/distDLS/

2 Setting Up Your Environment

2.1 Install DLStudio

Download version 2.5.5 of your instructor's DLStudio platform. Do NOT use `pip install` as that would not give you the Examples directory. The main documentation page is:

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

2.2 Install LVIS API

The LVIS API uses the same structure as COCO's `pycocotools`:

```
pip install lvis
# Or using conda:
conda install -c conda-forge lvis
```

2.3 Download LVIS Annotations

Visit the LVIS dataset page: https://www.lvisdataset.org/dataset

The annotation files are distributed as **.zip files**, not raw JSON:

```
# Method 1: Using wget (recommended)
wget https://dl.fbaipublicfiles.com/LVIS/lvis_v1_train.json.zip
wget https://dl.fbaipublicfiles.com/LVIS/lvis_v1_val.json.zip

unzip lvis_v1_train.json.zip -d annotations/
unzip lvis_v1_val.json.zip -d annotations/

# Method 2: Using aria2 (faster, more reliable)
apt-get install aria2  # On Ubuntu/Colab
aria2c -x 16 -s 16 https://dl.fbaipublicfiles.com/LVIS/
    lvis_v1_train.json.zip
unzip lvis_v1_train.json.zip -d annotations/
```

2.4 Download COCO 2017 Images

LVIS uses COCO images. Download from: https://cocodataset.org/#download

Download:

- **2017 Train images** (˜18 GB) - Required for this homework
- **2017 Val images** (˜1 GB) - Optional, useful for testing your pipeline

Storage Requirement: Plan for approximately **30 GB** of disk space after extraction (COCO train images: ˜25–27 GB extracted, LVIS annotations: ˜600 MB, plus workspace for your datasets).

Download Time: The training images may take 1–2 hours depending on connection speed. It took about 30 minutes on Colab.

2.5 Verify Installation

Test your LVIS installation:

```
from lvis import LVIS

# Load LVIS annotations
lvis_api = LVIS('path/to/lvis_v1_train.json')

# Get category count
num_cats = len(lvis_api.get_cat_ids())
print(f"LVIS has {num_cats} categories")  # Should print 1203

all_cats = lvis_api.load_cats(lvis_api.get_cat_ids())
cat_name_to_id = {cat['name']: cat['id'] for cat in all_cats}

cat_id = cat_name_to_id['dog']
```

```
print(f"Dog category ID: {cat_id}")

# Get images containing dogs
ann_ids = lvis_api.get_ann_ids(cat_ids=[cat_id])
anns = lvis_api.load_anns(ann_ids)
img_ids = list(set([ann['image_id'] for ann in anns]))
print(f"Images with dogs: {len(img_ids)}")
```

If this runs without errors, you are ready to proceed.

2.6 Understanding LVIS Structure

LVIS organizes its 1,203 categories into three frequency groups based on annotation counts in the training set:

- **Rare:** Categories with fewer than 10 training images (472 categories)
- **Common:** Categories with 10–100 training images (461 categories)
- **Frequent:** Categories with more than 100 training images (270 categories)

For this homework: You will select 5 categories from the **Common** or **Frequent** groups to ensure you have enough images for training.

2.7 LVIS API Basics

The LVIS API follows a similar pattern to COCO's API, but with some important differences. Here are the essential operations:

```
from lvis import LVIS

# Initialize API
lvis = LVIS('path/to/lvis_v1_train.json')

# Get all categories and their frequencies
all_cats = lvis.load_cats(lvis.get_cat_ids())

# Example: Explore categories
for cat in all_cats[:10]:  # First 10
    print(f"{cat['name']}: {cat['frequency']} frequency, "
          f"{cat['instance_count']} instances")

# IMPORTANT: Get images for a category via annotations
# (LVIS API differs from COCO here)
cat_name = 'dog'
cat_name_to_id = {cat['name']: cat['id'] for cat in all_cats}
```

```
cat_id = cat_name_to_id[cat_name]

# Get all annotations for this category
ann_ids = lvis.get_ann_ids(cat_ids=[cat_id])
anns = lvis.load_anns(ann_ids)

# Extract unique image IDs
img_ids = list(set([ann['image_id'] for ann in anns]))

print(f"Found {len(img_ids)} images containing {cat_name}")

# Load image info
img_info = lvis.load_imgs([img_ids[0]])[0]

# IMPORTANT: LVIS uses 'coco_url', not 'file_name'
print(f"Image URL: {img_info['coco_url']}")
# Extract filename: http://images.cocodataset.org/train2017
    /000000397133.jpg
filename = img_info['coco_url'].split('/')[-1]
```

3 Programming Tasks (100 Points Total)

3.1 Task 1: DLStudio Exploration (20 points)

Objective: Install DLStudio, run its CIFAR-10 CNN example, and demonstrate that you understand how `torch.nn` is used to construct and train a convolutional neural network.

3.1.1 Running the CIFAR-10 Example (8 points)

1. Navigate to the DLStudio Examples directory.
2. Execute the CIFAR-10 example:

   ```
   python playing_with_cifar10.py
   ```

   This script demonstrates:

   - CNN architecture definition using `torch.nn.Module`
   - A training loop with batch processing
   - Loss curve visualization
   - Accuracy calculation

3. Run the script to completion. Save the training loss curve and the final accuracy output.

Report:

- Include the loss curve plot produced by the script
- Report the final training and validation accuracy
- Confirm your DLStudio version (2.5.5)

3.1.2 Understanding the Network Architecture (12 points)

Study the network architectures in DLStudio's `ExperimentsWithCIFAR` inner class (see [1] Page 25). The `Net` and `Net2` examples show standard CNN patterns.

Open the script in one window while you execute it in another. Make changes to the script and observe what happens to the results, as recommended in the DLStudio lecture [1].

Report the following:

1. For the `Net` (or `Net2`) architecture used in the CIFAR-10 example:
   - List every layer and its type (Conv2d, MaxPool2d, Linear, etc.)
   - For each convolutional layer: state the input channels, output channels, kernel size, and padding
   - For each linear (fully connected) layer: state the input and output dimensions
   - Trace the spatial dimensions of the feature maps through the network (e.g., $32 \times 32 \rightarrow 16 \times 16 \rightarrow \ldots$)
2. Explain in your own words (2–3 sentences each):
   - What does `nn.Conv2d` do and what are its key parameters?
   - What is the role of `nn.MaxPool2d` in the network?
   - Why is a `view()` or `flatten()` call needed before the fully connected layers?
   - What does `nn.CrossEntropyLoss` compute and why is it appropriate for classification?
3. Compute the total number of trainable parameters in the network. Show your calculation for at least two layers (one convolutional, one fully connected).

Grading:

- Successful execution with loss curve and accuracy (8 points)
- Architecture analysis and `torch.nn` explanations (12 points)

3.2 Task 2: Dataset Creation from LVIS (20 points)

Your second task is to create three classification datasets from LVIS annotations. You will use the same 5 categories across all three datasets.

3.2.1 Selecting Your 5 Categories (5 points)

Selection Criteria:

- Choose categories from **Common** or **Frequent** frequency groups
- Ensure each category has at least 500 images in LVIS training set
- Select visually distinct categories (easier for classification)
- Avoid very similar categories (e.g., "dog" and "dog_beagle")

Implementation:

```
from lvis import LVIS
lvis = LVIS('path/to/lvis_v1_train.json')
all_cats = lvis.load_cats(lvis.get_cat_ids())

# Group by frequency
freq_groups = {'r': [], 'c': [], 'f': []}
for cat in all_cats:
    if cat['frequency'] in freq_groups:
        freq_groups[cat['frequency']].append({
            'name': cat['name'],
            'id': cat['id'],
            'count': cat['instance_count']
        })

# Create a mapping for display purposes
freq_display_map = {'f': 'frequent', 'c': 'common', 'r': 'rare'
    }

for freq_code in ['f', 'c']:
    display_name = freq_display_map[freq_code]
    sorted_cats = sorted(freq_groups[freq_code],
                         key=lambda x: x['count'],
                         reverse=True)
    for i, cat_info in enumerate(sorted_cats[:30], 1):
        print(f"  {i:2d}. {cat_info['name']:20s}: {cat_info['
    count']:5d} instances")
```

Report in PDF:

- List your 5 selected categories
- For each, report: frequency group, instance count in LVIS
- Briefly explain why you chose these categories

3.2.2 Dataset 1: Single-Instance Images (5 points)

Objective: Extract images containing exactly ONE instance of the target category. You also need to make sure the same image in not included in more than one of your selected catergories while creating the dataset.

Filtering Criteria:

- Image must contain exactly 1 instance of target category (Hint: Some categories like banana or apple may make this harder)
- Instance area $\geq 5\%$ of total image area (reasonably sized)
- Image may contain other objects from different categories
- Prefer instances that are not heavily occluded

Target Size: 400 training + 100 validation images per class

Implementation Template:

```
def create_single_instance_dataset(lvis, category_name,
    category_info, min_area_ratio=0.05, max_images=500):
    """
    Extract images with exactly one instance of category.

    Args:
        lvis: LVIS API object
        category_name: Target category (e.g., 'person')
        category_info: Dictionary with pre-loaded category
    information
        min_area_ratio: Minimum object area relative to image
        max_images: Maximum images to extract

    Returns:
        List of image data dictionaries
    """
    # Get category ID from our pre-built dictionary
    if category_name not in category_info:
        raise ValueError(f"Category '{category_name}' not found
     in category_info")
```

```
18
19      cat_id = category_info[category_name]['cat_id']
20      img_ids = category_info[category_name]['img_ids']
21
22      single_instance_data = []
23
24      for img_id in tqdm(img_ids, desc=f"Single-instance: {
        category_name}"):
25          # Load image info
26          img_info = lvis.load_imgs([img_id])[0]
27          img_area = img_info['width'] * img_info['height']
28
29          # Load all annotations for this image
30          ann_ids = lvis.get_ann_ids(img_ids=[img_id])
31          anns = lvis.load_anns(ann_ids)
32
33          # Filter for target category
34          target_anns = [a for a in anns if a['category_id'] ==
        cat_id]
35
36          # Check: exactly 1 instance
37          if len(target_anns) != 1:
38              continue
39
40          ann = target_anns[0]
41
42          # Check: sufficient size
43          if ann['area'] / img_area < min_area_ratio:
44              continue
45
46          single_instance_data.append({
47              'img_id': img_id,
48              'img_info': img_info,
49              'annotation': ann,
50              'category': category_name
51          })
52
53          if len(single_instance_data) >= max_images:
54              break
55
56      return single_instance_data
57
58  SELECTED_CATEGORIES = [
59      'shirt',
60      'plate',
61      'person',
62      'tomato',
63      'pizza',
64  ]
```

```
65
66 cat_name_to_id = {cat_info['name']: cat_info['id'] for cat_info
       in all_cats}
67
68 # Store category info
69 category_info = {}
70
71 for cat_name in SELECTED_CATEGORIES:
72     if cat_name in cat_name_to_id:
73         cat_id = cat_name_to_id[cat_name]
74
75     cat_info = lvis.load_cats([cat_id])[0]
76     ann_ids = lvis.get_ann_ids(cat_ids=[cat_id])
77     anns = lvis.load_anns(ann_ids)
78
79     img_ids = list(set([ann['image_id'] for ann in anns]))
80
81     category_info[cat_name] = {
82         'cat_id': cat_id,
83         'cat_info': cat_info,
84         'img_ids': img_ids,
85         'num_annotations': len(anns)
86     }
87
88 total_images = sum(len(info['img_ids']) for info in
     category_info.values())
89 total_annotations = sum(info['num_annotations'] for info in
     category_info.values())
90 print(f"  Total unique images: {total_images:,}")
91 print(f"  Total annotations: {total_annotations:,}")
92
93 # Test with one category
94 test_single = create_single_instance_dataset(
95     lvis,
96     SELECTED_CATEGORIES[0],
97     category_info,
98     max_images=10
99 )
100 print(f"\nTest: Found {len(test_single)} single-instance images
       for '{SELECTED_CATEGORIES[0]}'")
```

3.2.3 Dataset 2: Multi-Instance Same Object (5 points)

Objective: Extract images containing 2 or more instances of the same category.

Filtering Criteria:

- Image must contain ≥ 2 instances of target category

- Combined area of all target instances $\geq 10\%$ of image area
- All instances should be reasonably visible

Target Size: 400 training + 100 validation images per class
Implementation Hint:

```
def create_multi_same_dataset(lvis, category_name,
    category_info,
                              min_instances=2,
                              min_total_area_ratio=0.10):
    """Extract images with multiple instances of same category.
    """
    # Similar structure to single-instance
    # Key difference: len(target_anns) >= min_instances
    # Check: sum of areas >= min_total_area_ratio
    pass  # Your implementation
```

3.2.4 Dataset 3: Multi-Instance Different Objects (5 points)

Objective: Extract images containing the target category and at least one other category from your selected 5.
Filtering Criteria:

- Image contains target category
- Image also contains ≥ 1 instance from a different category in your selected 5
- Target category instances' total area $\geq 5\%$ of image area

Target Size: 400 training + 100 validation images per class

```
def create_multi_different_dataset(lvis, selected_categories,
                                   category_info,
                                   min_area_ratio=0.01):
    """Extract images with multiple DIFFERENT categories."""
    # Collect all images across all selected categories
    # Key difference: len(cats_present) >= 2 different
    categories
    # Assign image to: max(category_areas)
    # Check: primary_area / img_area >= min_area_ratio
    pass  # Your implementation
```

Implementation Note: Each image should be assigned to exactly one of your 5 classes based on which category has the largest total instance area in that image. If you forget to do this, your classification accuracy will be affected.

3.3 Task 3: Image Processing and Storage (10 points)

Objective: Save extracted images in an organized directory structure with proper preprocessing.

Directory Structure:

```
lvis_datasets/
+-- single_instance/
|   +-- train/
|   |   +-- category1/
|   |   +-- category2/
|   |   +-- ...
|   +-- val/
|       +-- (same classes)
+-- multi_instance_same/
|   +-- (same structure)
+-- multi_instance_different/
    +-- (same structure)
```

Image Processing Requirements:

- Resize all images to 64×64 pixels
- Use bicubic or bilinear interpolation
- Maintain RGB format (3 channels)

Data Augmentation (if needed):

If you cannot find enough images for certain categories, apply data augmentation. Be careful to separate your training and validation sets *before* augmenting to prevent data leakage.

Report Requirements:

- Document your final dataset sizes in a table
- Note any data augmentation applied
- Include sample images (3 per class per dataset type)

3.4 Task 4: CNN Architecture Implementation (15 points)

Objective: Implement LVISNet, a convolutional neural network for 5-class classification on 64×64 images using `torch.nn.Module`.

Network Architecture - LVISNet:

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class LVISNet(nn.Module):
    """
    CNN for LVIS-based image classification.

    Input: (batch_size, 3, 64, 64) RGB images
    Output: (batch_size, 5) class logits
    """

    def __init__(self, num_classes=5):
        super(LVISNet, self).__init__()

        # Convolutional Block 1
        # Input: 3 x 64 x 64
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.pool1 = nn.MaxPool2d(2, 2)
        # Output: 32 x 32 x 32

        # Convolutional Block 2
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding
    =1)
        self.bn2 = nn.BatchNorm2d(64)
        self.pool2 = nn.MaxPool2d(2, 2)
        # Output: 64 x 16 x 16

        # Convolutional Block 3
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding
    =1)
        self.bn3 = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(2, 2)
        # Output: 128 x 8 x 8

        # Calculate flattened size: 128 * 8 * 8 = 8192

        # Fully Connected Layers
        self.fc1 = nn.Linear(128 * 8 * 8, 256)
        self.dropout1 = nn.Dropout(0.5)

        self.fc2 = nn.Linear(256, 128)
        self.dropout2 = nn.Dropout(0.5)

        self.fc3 = nn.Linear(128, num_classes)

    def forward(self, x):
        # Convolutional Block 1
```

```
        x = self.conv1(x)
        x = self.bn1(x)
        x = F.relu(x)
        x = self.pool1(x)

        # Convolutional Block 2
        x = self.conv2(x)
        x = self.bn2(x)
        x = F.relu(x)
        x = self.pool2(x)

        # Convolutional Block 3
        x = self.conv3(x)
        x = self.bn3(x)
        x = F.relu(x)
        x = self.pool3(x)

        # Flatten
        x = x.view(x.size(0), -1)

        # Fully Connected Layers
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout1(x)

        x = self.fc2(x)
        x = F.relu(x)
        x = self.dropout2(x)

        x = self.fc3(x)

        return x

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

# Create and inspect network
net = LVISNet(num_classes=5)
print(f"Network: {net}")
print(f"Total parameters: {net.count_parameters():,}")

# Test with dummy input
dummy_input = torch.randn(4, 3, 64, 64)  # batch of 4
output = net(dummy_input)
print(f"Input shape: {dummy_input.shape}")
print(f"Output shape: {output.shape}")  # Should be (4, 5)
```

Architecture Analysis (Required for Report):
Calculate and document:

1. Total number of trainable parameters
2. Output dimensions after each layer:
    - After conv1 + pool1: ?
    - After conv2 + pool2: ?
    - After conv3 + pool3: ?
    - After flatten: ?
    - After fc1: ?
    - After fc2: ?
    - After fc3: ?
3. Number of parameters in each layer (conv1, conv2, conv3, fc1, fc2, fc3)

Grading:

- Correct architecture implementation (8 points)
- Proper initialization (2 points)
- Parameter counting and dimension analysis (3 points)
- Code documentation (2 points)

3.5 Task 5: Custom Dataset and DataLoader (10 points)

Objective: Implement a PyTorch `Dataset` and `DataLoader` for your LVIS-based datasets.

Report Requirements:

- Verify dataset loading works correctly
- Report number of images per class
- Show sample transformed images from training set
- Document any data augmentation applied

3.6 Task 6: Training Implementation (15 points)

Objective: Implement and execute the training loop for all three datasets.

Training Configuration:

```
import torch
import torch.nn as nn
import torch.optim as optim

# Device setup
device = torch.device('cuda' if torch.cuda.is_available() else
    'cpu')
print(f"Using device: {device}")

# Initialize network
net = LVISNet(num_classes=5).to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=1e-3, betas=(0.9,
    0.99))

# Training parameters
epochs = 7
batch_size = 32
```

Training Loop:

```
from tqdm import tqdm

# Storage for metrics
train_losses = []
train_accuracies = []
val_losses = []
val_accuracies = []

for epoch in range(epochs):
    print(f"\nEpoch {epoch+1}/{epochs}")
    print("-" * 50)

    # Training phase
    net.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for inputs, labels in tqdm(train_loader, desc="Training"):
        inputs, labels = inputs.to(device), labels.to(device)

        # Forward pass
```

```
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)

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

        # Statistics
        running_loss += loss.item()
        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()

    epoch_train_loss = running_loss / len(train_loader)
    epoch_train_acc = 100. * correct / total
    train_losses.append(epoch_train_loss)
    train_accuracies.append(epoch_train_acc)

    print(f"Train Loss: {epoch_train_loss:.4f}, "
          f"Train Acc: {epoch_train_acc:.2f}%")

    # Validation phase
    net.eval()
    val_running_loss = 0.0
    val_correct = 0
    val_total = 0

    with torch.no_grad():
        for inputs, labels in tqdm(val_loader, desc="Validation
    "):
            inputs, labels = inputs.to(device), labels.to(
    device)
            outputs = net(inputs)
            loss = criterion(outputs, labels)

            val_running_loss += loss.item()
            _, predicted = outputs.max(1)
            val_total += labels.size(0)
            val_correct += predicted.eq(labels).sum().item()

    epoch_val_loss = val_running_loss / len(val_loader)
    epoch_val_acc = 100. * val_correct / val_total
    val_losses.append(epoch_val_loss)
    val_accuracies.append(epoch_val_acc)

    print(f"Val Loss: {epoch_val_loss:.4f}, "
          f"Val Acc: {epoch_val_acc:.2f}%")

```

```python
# Save model
torch.save(net.state_dict(), 'lvisnet_single_instance.pth')
print("\nTraining complete! Model saved.")
```

Train on All Three Datasets:
You must train LVISNet separately on:

1. Single-instance dataset
2. Multi-instance same object dataset
3. Multi-instance different objects dataset

For each dataset:

- Train for 7 epochs with the same hyperparameters
- Save training/validation loss and accuracy curves
- Save final model weights
- Record training time

Grading:

- Training loop implementation (8 points)
- Successful training on all 3 datasets (4 points)
- Proper metric tracking (3 points)

3.7 Task 7: Confusion Matrix (10 points)

Objective: Implement confusion matrix calculation from scratch (no sklearn for computation, but visualization libraries like matplotlib/seaborn are allowed).

Theory: A confusion matrix C is an $n \times n$ matrix where C_{ij} represents the number of samples with true label i that were predicted as label j.

$$C_{ij} = \sum_{k=1}^{N} \mathbb{1}[\text{true}_k = i \text{ and } \text{pred}_k = j] \tag{1}$$

Visualization:

```python
import matplotlib.pyplot as plt
import seaborn as sns

def plot_confusion_matrix(cm, class_names, title, save_path=None):
    """Plot confusion matrix with matplotlib/seaborn."""
    plt.figure(figsize=(10, 8))

    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=class_names,
                yticklabels=class_names,
                cbar_kws={'label': 'Count'})

    plt.title(title, fontsize=14, fontweight='bold')
    plt.ylabel('True Label', fontsize=12)
    plt.xlabel('Predicted Label', fontsize=12)
    plt.tight_layout()

    if save_path:
        plt.savefig(save_path, dpi=300, bbox_inches='tight')
    plt.show()

# Plot confusion matrix
plot_confusion_matrix(
    cm, class_names,
    title='Single Instance Dataset - Confusion Matrix',
    save_path='cm_single_instance.png'
)
```

Requirements:

- Implement confusion matrix calculation without sklearn (4 points)
- Compute and visualize for all 3 datasets (4 points)
- Calculate and report per-class accuracy and overall accuracy (2 points)

Report: Include all three confusion matrices in your report. Briefly note which classes are most frequently confused and whether this differs across the three datasets.

4 Bonus Tasks (20 Points)

Implement two deeper variants of LVISNet:

1. **LVISNet-Deep5:** 5 convolutional blocks (vs 3 in baseline)

2. **LVISNet-Deep10:** 10 convolutional blocks

Requirements:

- Train both on multi-instance different dataset only
- Use same training configuration as baseline
- Report:
  - Training curves (baseline vs Deep5 vs Deep10)
  - Confusion matrices for all three
  - Parameter count comparison
  - Training time per epoch
- Analysis: Does depth help? What challenges arise with deeper networks?

5 Submission Instructions

5.1 What to Submit

1. **PDF Report** (submit to Gradescope, mark pages):
   - DLStudio Exploration:
     - Loss curve from CIFAR-10 example
     - Architecture analysis and `torch.nn` explanations
     - Parameter calculations
   - Dataset Creation:
     - Category selection and justification
     - Filtering criteria for each dataset type
     - Dataset statistics table
     - Sample images (3×5 grid per dataset type)
   - Network Architecture:
     - LVISNet description
     - Parameter count and dimension analysis
   - Training Results:
     - Training curves for all datasets

    - Confusion matrices (3 total, 6 if normalized)
    - Results summary table
  - Brief discussion of results and any challenges encountered
  - (Optional) Bonus: Results and analysis

2. **Code Files** (submit as zip - see Section 5.2 for exact naming and signature requirements):
   - `dataset_creation.py`: LVIS dataset extraction
   - `models.py`: LVISNet implementation
   - `train.py`: Training loop
   - `evaluate.py`: Evaluation and confusion matrix
   - `visualize.py`: Plotting functions
   - `requirements.txt`: Dependencies
   - `README.md`: Instructions to run

5.2 Autograder Interface Requirements

Your code will be partially graded by an autograder. 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. Read this section carefully.

5.2.1 Required File Names

Submit exactly these file names (case-sensitive):

- `models.py`
- `dataset_creation.py`
- `train.py`
- `evaluate.py`

5.2.2 `models.py`

Must contain a class named `LVISNet`:

```python
import torch
import torch.nn as nn

class LVISNet(nn.Module):
    def __init__(self, num_classes=5):
        """
        Args:
            num_classes (int): Number of output classes.
    Default: 5.
        """
        super(LVISNet, self).__init__()
        # ... your layers ...

    def forward(self, x):
        """
        Args:
            x (torch.Tensor): Input tensor of shape (B, 3, 64,
    64).
        Returns:
            torch.Tensor: Output logits of shape (B,
    num_classes).
        """
        # ... your forward pass ...
        return x

    def count_parameters(self):
        """
        Returns:
            int: Total number of trainable parameters.
        """
        return sum(p.numel() for p in self.parameters()
                   if p.requires_grad)
```

The autograder will run:

```python
from models import LVISNet
net = LVISNet(num_classes=5)
out = net(torch.randn(4, 3, 64, 64))
assert out.shape == (4, 5)
assert isinstance(net.count_parameters(), int)
```

5.2.3 `evaluate.py`

Must contain these two functions:

```python
import numpy as np
import torch

```

```python
def compute_confusion_matrix(model, dataloader, num_classes, device):
    """
    Compute confusion matrix from scratch (no sklearn).

    Args:
        model (nn.Module): Trained model (already on device).
        dataloader (DataLoader): Validation/test DataLoader.
        num_classes (int): Number of classes.
        device (torch.device): Device to run inference on.

    Returns:
        np.ndarray: Confusion matrix of shape (num_classes, num_classes).
                    Entry cm[i][j] = number of samples with true label i
                    predicted as label j.
    """
    # ... your implementation (no sklearn) ...

def per_class_accuracy(cm):
    """
    Compute per-class accuracy from a confusion matrix.

    Args:
        cm (np.ndarray): Confusion matrix of shape (num_classes, num_classes).

    Returns:
        np.ndarray: Per-class accuracy array of shape (num_classes,).
                    Entry i = cm[i][i] / sum(cm[i]).
    """
    # ... your implementation ...
```

The autograder will run:

```python
from evaluate import compute_confusion_matrix, per_class_accuracy
import numpy as np

# Test per_class_accuracy with a known matrix
test_cm = np.array([[10, 2, 0],
                    [1, 8, 3],
                    [0, 1, 9]])
acc = per_class_accuracy(test_cm)
assert acc.shape == (3,)
assert np.isclose(acc[0], 10/12)
```

5.2.4 `train.py`

Must contain these two functions:

```python
import torch

def train_one_epoch(model, dataloader, criterion, optimizer, device):
    """
    Train the model for one epoch.

    Args:
        model (nn.Module): The network (already on device).
        dataloader (DataLoader): Training DataLoader.
        criterion: Loss function (e.g., nn.CrossEntropyLoss()).
        optimizer: Optimizer (e.g., optim.Adam).
        device (torch.device): Device for training.

    Returns:
        tuple: (avg_loss, accuracy) where
            avg_loss (float): Mean loss over all batches.
            accuracy (float): Classification accuracy as percentage (0-100).
    """
    # ... your implementation ...

def validate(model, dataloader, criterion, device):
    """
    Evaluate the model (no gradient computation).

    Args:
        model (nn.Module): The network (already on device).
        dataloader (DataLoader): Validation DataLoader.
        criterion: Loss function.
        device (torch.device): Device for evaluation.

    Returns:
        tuple: (avg_loss, accuracy) where
            avg_loss (float): Mean loss over all batches.
            accuracy (float): Classification accuracy as percentage (0-100).
    """
    # ... your implementation ...
```

The autograder will run:

```python
from train import train_one_epoch, validate
loss, acc = train_one_epoch(model, train_loader, criterion, optimizer, device)
assert isinstance(loss, float)
```

```
assert isinstance(acc, float)
```

5.2.5 dataset_creation.py

Must contain a module-level list and three functions:

```
from lvis import LVIS

# Module-level list - exactly 5 category name strings
SELECTED_CATEGORIES = [
    'your_category_1',
    'your_category_2',
    'your_category_3',
    'your_category_4',
    'your_category_5',
]

def create_single_instance_dataset(lvis, category_name, category_info,
                                   min_area_ratio=0.05,
                                   max_images=500):
    """
    Extract images with exactly one instance of category.

    Args:
        lvis (LVIS): LVIS API object.
        category_name (str): Target category name.
        category_info (dict): Pre-loaded category information dict
            where category_info[cat_name] contains at least
            'cat_id' (int) and 'img_ids' (list of int).
        min_area_ratio (float): Min object area / image area.
        max_images (int): Maximum number of images to return.

    Returns:
        list[dict]: Each dict has keys:
            'img_id' (int), 'img_info' (dict),
            'annotation' (dict), 'category' (str).
    """
    # ... your implementation ...

def create_multi_same_dataset(lvis, category_name, category_info,
                              min_instances=2,
                              min_total_area_ratio=0.10):
    """
    Extract images with multiple instances of same category.

```

```
    Args:
        lvis (LVIS): LVIS API object.
        category_name (str): Target category name.
        category_info (dict): Pre-loaded category information.
        min_instances (int): Minimum number of instances required.
        min_total_area_ratio (float): Min combined area / image area.

    Returns:
        list[dict]: Each dict has keys:
            'img_id' (int), 'img_info' (dict),
            'annotations' (list[dict]), 'category' (str).
    """
    # ... your implementation ...

def create_multi_different_dataset(lvis, selected_categories,
                                   category_info,
                                   min_area_ratio=0.01):
    """
    Extract images containing multiple different categories
    from your selected 5.

    Args:
        lvis (LVIS): LVIS API object.
        selected_categories (list[str]): List of 5 category names.
        category_info (dict): Pre-loaded category information.
        min_area_ratio (float): Min primary category area / image area.

    Returns:
        dict: Maps each category name (str) to a list[dict],
            where each dict has keys:
            'img_id' (int), 'img_info' (dict),
            'annotations' (list[dict]), 'assigned_category' (str).
    """
    # ... your implementation ...
```

The autograder will run:

```
from dataset_creation import SELECTED_CATEGORIES
assert isinstance(SELECTED_CATEGORIES, list)
assert len(SELECTED_CATEGORIES) == 5
assert all(isinstance(c, str) for c in SELECTED_CATEGORIES)
```

5.2.6 Summary Table

File	Required Names	Autograder Tests
`models.py`	`LVISNet` class	Forward pass shape, param count
`evaluate.py`	`compute_confusion_matrix`, `per_class_accuracy`	CM shape, known-input test, accuracy values
`train.py`	`train_one_epoch`, `validate`	Return types (float, float)
`dataset_creation.py`	`SELECTED_CATEGORIES`, 3 creation functions	List length = 5, function signatures

Important Notes:

- Do **not** rename these files or functions. The autograder imports them by exact name.
- Do **not** put executable code (training loops, print statements, etc.) at the module level. Guard any scripts with `if __name__ == '__main__':` blocks so that imports do not trigger side effects.
- You may add additional helper functions to any file, but the required functions above must exist with exactly the specified signatures.
- Default argument values must match those shown above.

5.3 Code Quality Requirements

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

5.4 Report Formatting

- Include outputs directly next to corresponding code sections
- Use figures and tables with captions

- Number equations, figures, and tables
- Use proper citations if referencing papers

5.5 Reproducibility

Include this at the start of your code:

```python
import random
import numpy as np
import torch

# Set random seeds for reproducibility
random.seed(42)
np.random.seed(42)
torch.manual_seed(42)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(42)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
```

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
- Previous years' solutions are for reference only

6 Common Problems and Solutions

This section addresses the most frequent technical issues students encounter. Check here first before posting to Piazza!

6.1 Download Issues

Problem: LVIS annotations won't download or URLs don't work
Solutions:

1. **Use correct URLs:** Files are now `.zip` format, not raw JSON

```
# CORRECT URLs:
wget https://dl.fbaipublicfiles.com/LVIS/lvis_v1_train.
    json.zip
wget https://dl.fbaipublicfiles.com/LVIS/lvis_v1_val.json.
    zip

# Then unzip:
unzip lvis_v1_train.json.zip -d annotations/
unzip lvis_v1_val.json.zip -d annotations/
```

2. **Try aria2c instead of wget** (more reliable for large files):

```
apt-get install aria2  # Ubuntu/Colab
aria2c -x 16 -s 16 https://dl.fbaipublicfiles.com/LVIS/
    lvis_v1_train.json.zip
```

3. **Manual download:** Visit https://www.lvisdataset.org/dataset, download manually, upload to Colab

4. **Verify file integrity:**

```
# Train file should be ~560 MB uncompressed
ls -lh annotations/lvis_v1_train.json
# Should show ~560M or larger
```

6.2 LVIS API Errors

Problem: `KeyError: 'file_name'`
Solution: LVIS uses `'coco_url'`, not `'file_name'`

```
# Fix:
filename = img_info['coco_url'].split('/')[-1]
```

Problem: `TypeError: get_img_ids() got unexpected keyword argument 'cat_ids'`
Solution: Get images via annotations (LVIS API differs from COCO)

```
# Correct approach:
ann_ids = lvis.get_ann_ids(cat_ids=[cat_id])
anns = lvis.load_anns(ann_ids)
img_ids = list(set([ann['image_id'] for ann in anns]))
```

Problem: Category not found (e.g., "car" not found)
Solution: Use exact LVIS names with parentheses:

```
# WRONG:
'car'  # Not found!

```

```
# CORRECT:
'car_(automobile)'  # Found!

# Search first to find exact name:
matches = [c for c in all_cats if 'car' in c['name'].lower()]
```

6.3 Data Scarcity Issues

Problem: Only 50–200 images per category found
Solutions:

1. **Relax area threshold:**

```
# Change from:
min_area_ratio=0.05  # Too strict

# To:
min_area_ratio=0.01  # Much more lenient
```

2. **Apply data augmentation:** Acceptable to augment images and also consider using the validation set from COCO.

3. **Choose different categories:** Select from categories with >10,000 instances

4. **Use 80/20 split:** If fewer than 500 images total, don't force 400/100 split

Problem: Multi-different has <20 images per category
Solutions:

1. Multi-different is inherently sparse

2. **Choose co-occurring categories:** person + chair + table + cup + bottle appear together

3. **Document in report:** Explain the sparsity and augmentation strategy

6.4 Training Issues

Problem: Dimension mismatch errors
Solution: Add debug prints in forward():

```python
def forward(self, x):
    print(f"Input: {x.shape}")  # Should be (B, 3, 64, 64)
    x = self.conv1(x)
    print(f"After conv1: {x.shape}")  # Track shape changes
    # ... continue for each layer
```

Problem: CUDA out of memory
Solution: Reduce batch size:

```python
batch_size = 16  # or even 8 if still failing
```

Problem: Training is extremely slow
Solutions:

1. Verify GPU is being used: `print(torch.cuda.is_available())`
2. Reduce `num_workers` in DataLoader (try 0 or 2)
3. Use Google Colab if training on CPU locally

Problem: Low accuracy (30–40%)
Diagnostics:

1. Check if loss is decreasing (most important indicator)
2. Compare to random baseline (20% for 5 classes)
3. Check train vs val accuracy gap ($> 20\%$ = overfitting)
4. Verify image normalization is applied

6.5 Code Issues

Problem: Images not loading from COCO directory
Solutions:

1. Verify COCO directory path:

```python
import os
print(os.path.exists('train2017'))  # Should be True
# Check a specific image:
print(os.path.exists('train2017/000000397133.jpg'))
```

2. Check extracted properly:

```
# Should have ~118k images
ls train2017/ | wc -l
```

3. Make sure you unzipped, not just downloaded zip:

```
unzip train2017.zip
```

Problem: Confusion matrix all zeros
Solutions:

1. Verify model is in eval mode: `model.eval()`
2. Check predictions are being made correctly
3. Ensure label indices match (0–4 for 5 classes)
4. Verify dataloader isn't empty:

```
for imgs, labels in val_loader:
    print(f"Batch: {imgs.shape}, Labels: {labels}")
    break  # Check first batch
```

7 Frequently Asked Questions (FAQ)

Q: How long will the COCO images take to download?
A: The 2017 Train images are 18GB. Depending on your internet speed, expect 1–3 hours. On Colab it took about 30 minutes.

Q: Can I use COCO 2017 Val images to test my pipeline first?
A: Yes. The Val set is only 1GB and perfect for testing. Once your code works, switch to the Train set for the actual homework. You may include Val set if you face data scarcity issues.

Q: What if I cannot find 500 images for a category?
A: This is common with LVIS. Solutions in order:

1. **Relax filtering criteria:** Use `min_area_ratio=0.01` instead of 0.05 (Just know this will lead to bad classification accuracy)
2. **Apply data augmentation:** Augmenting is acceptable as well as using the Val set from COCO. Be careful not to mix up your training and validation set data.
3. **Choose different category:** Select from categories with >10,000 instances
4. **Document your approach:** Explain thresholds and augmentation in report

Q: Should I use the same 5 categories for all three datasets?

A: Yes. Use the same 5 categories across all three dataset types. This ensures fair comparison.

Q: My LVIS categories do not match my selected names.

A: LVIS category names are sometimes specific. Use the exploration code to find exact names. For example, "bicycle" might be "bicycle" or "bike". Check carefully.

Q: How do I handle images with both target and non-target objects?

A: For single-instance and multi-same datasets, images can contain other objects from non-selected categories-that is fine. For multi-different dataset, other objects must be from your selected 5 categories. Be careful selecting your categories here because it will impact the number of images you obtain.

Q: For multi-different dataset, how do I decide which class an image belongs to?

A: Assign each image to the category with the **largest total instance area**. For example, if an image contains 3 dogs (total area 5000 $px \times px$) and 2 cars (total area 8000 $px \times px$), assign it to the "car" class.

Q: What does "$area \geq 5\%$ of image" mean mathematically?

A: If annotation area is A_{obj} and image dimensions are $W \times H$:

$$\frac{A_{obj}}{W \times H} \geq 0.05 \tag{2}$$

The LVIS API provides A_{obj} directly in the annotation's 'area' field.

Q: Why resize to 64×64? Isn't that very small?

A: Yes, but it keeps training fast enough to run on a laptop. For research or production you would use larger images (224×224 or more), but 64×64 is sufficient for learning the pipeline. If you have access to more compute, feel free to experiment with higher resolution.

Q: Should padding be 'same' or 'valid' in convolutions?

A: The provided LVISNet uses `padding=1` with `kernel_size=3`, which maintains spatial dimensions. This is standard practice.

Q: What does `view(x.size(0), -1)` do?

A: It flattens the tensor. `x.size(0)` is the batch size, `-1` means "infer the remaining dimension". It converts (B, C, H, W) to $(B, C \times H \times W)$ for the fully connected layers.

Q: My network gives dimension mismatch errors. How to debug?

A: Add print statements in forward():

```python
def forward(self, x):
    print(f"Input: {x.shape}")
    x = self.conv1(x)
    print(f"After conv1: {x.shape}")
    # ... continue for each layer
```

Find where dimensions break and fix the mismatch.

Q: What learning rate should I use?

A: The homework specifies `lr=1e-3` (0.001) with Adam optimizer. This is a good default. If training is unstable, try 1e-4. If too slow, try 3e-3.

Q: Only 7 epochs seems short. Why not more?

A: With a small dataset (˜2,000 training images) and a simple task, 7 epochs is sufficient to see trends. You are welcome to train longer, but it is not required.

Q: My validation accuracy is lower than training. Is this bad?

A: No, this is expected (it is called the generalization gap). If validation is much lower ($> 20\%$ gap), you might be overfitting. Dropout and data augmentation help.

Q: How do I compute confusion matrix element C_{ij}?

A: For each prediction:

```python
true_label = labels[k].item()  # True class
pred_label = predictions[k].item()  # Predicted class
confusion_matrix[true_label][pred_label] += 1
```

Repeat for all test samples.

Q: Can I use sklearn or seaborn for confusion matrix?

A: **For computation:** No sklearn allowed. You must implement the confusion matrix calculation from scratch (see Task 7).

For visualization: Yes. You can use matplotlib, seaborn, or any visualization library to plot the matrix.

Q: My accuracy is only 30–40%. Is my code broken?

A: Not necessarily. With 5 classes, random guessing gives 20%. Check whether loss is decreasing-if so, the network is learning. **Focus on relative comparisons** between dataset types, not absolute accuracy. Depending on your category choices, 30–40% may be reasonable.

Q: Should I include code in my report or just results?

A: Include key code snippets (10–20 lines) showing important parts (network definition, training loop structure, confusion matrix calculation). Do not paste entire files-that goes in the zip submission.

Q: Can I work in a group?

A: No. This is individual work. You can discuss concepts with classmates, but all code and written work must be your own.

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

A: Yes. With 64×64 images and LVISNet's modest size, CPU training is feasible (maybe 10–15 min per epoch vs 1–2 min on GPU). Alternatively, use Google Colab's free GPU.

Q: I selected popular categories like 'car' or 'train' but get "Category not found" errors.

A: LVIS uses specific naming conventions with parentheses and underscores. Examples:

- **car** → `car_(automobile)`
- **train** → `train_(railroad_vehicle)`
- **orange** → `orange_(fruit)`
- **bus** → `bus_(vehicle)`

Solution: Search for your category first:

```
# Search for categories containing "car"
all_cats = lvis.load_cats(lvis.get_cat_ids())
search_term = "car"

matches = [cat for cat in all_cats
           if search_term.lower() in cat['name'].lower()]

for cat in matches:
    print(f"{cat['name']}: {cat['instance_count']} instances")
```

Use the **exact name** shown (including parentheses and underscores).

Q: I cannot find 500 images per category even with popular categories. What's wrong?

A: The filtering criteria in the homework code examples (5% area ratio) are **illustrative but often too strict**. However, you should remember that the smaller the instance is on the image the harder it will be for your primitive classifier to classify it.

```
# Homework shows: min_area_ratio=0.05 (5% of image)
# This is TOO STRICT for most categories

single_data = create_single_instance_dataset(
    lvis, category_name,
    min_area_ratio=0.01  # 0.1% - much more lenient!
)

multi_same_data = create_multi_same_dataset(
```

```
    lvis, category_name,
    min_instances=2,
    min_total_area_ratio=0.01  # 1% combined
)

multi_diff_data = create_multi_different_dataset(
    lvis, selected_categories,
    min_area_ratio=0.01  # 1%
)
```

Q: Even with relaxed criteria, I only get 100–300 images per category. What now?

A: **Use data augmentation** to reach 500 per category. Remember to separate the training and validation sets before augmenting to prevent data leakage.

Q: Should I split 400 train / 100 val if I only have 200 images total?

A: **No.** Use **80/20 split** when you have fewer images:

```
# IF you have 500+ images per category:
train_size = 400
val_size = 100

# IF you have fewer (after augmentation):
total = len(dataset)
train_size = int(0.8 * total)  # 80%
val_size = total - train_size   # 20%
```

Document your actual split sizes in your report.

Q: Can I use pretrained models like ResNet?

A: No. You must implement and train LVISNet from scratch. The point is to understand `torch.nn` and CNN training, not to achieve state-of-the-art accuracy.

References

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

[2] 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

[3] Avinash Kak, *A First Introduction to Torch.nn for Designing Deep Networks and to DLStudio for Experimenting with Them.* Available at: https://engineering.purdue.edu/kak/pdf-kak/TorchnnAndDLStudio.pdf