# ECE 60146: Homework 7

James Allan (jlallan@purdue.edu)

March 16, 2025

## 1 Creating Your Own Multi-Instance Object Localization Dataset

To generate the multi-instance object localization dataset, I started by filtering out all images that contained at least one of any of the following classes – cat, bus, and pizza – with a total pixel area of greater than 40000 square-pixels, which ensures that any labeled objects are in the foreground of the image. This yielded 3954 images for the training dataset and 2059 images for the validation dataset respectively. I then filtered out duplicate images and standarized the image size of $256 \times 256$. The latter step also required me to resize the bounding box information for any objects contained within a resized image. To track the updated annotations, I created a new COCO manifest annotations file for my specific, resized subset of the original COCO data. These steps are shown in Lines 33–68 of Listing 1.

Separately, to confirm the accuracy of my dataset, I wrote a plotting routine to generate a $3 \times 3$ grid of example images, with three images per class, as shown in Figure 1. The code for plotting is shown in Listing 2.

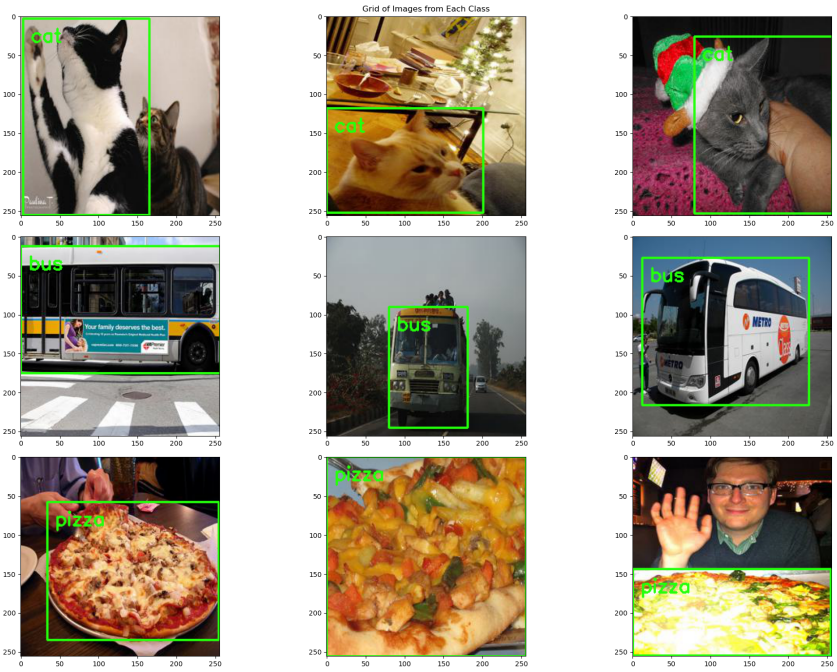


Figure 1: Example images and bounding boxes (green) from each of the three classes: cat (row 1), bus (row 2), and pizza (row 3)

Listing 1: Contents of `extract_images_coco.py`

```
1 from pycocotools.coco import COCO
2 from PIL import Image
```

```
3  import json
4  import os
5
6  def save_image(img_dir, out_dir, img_name, new_size):
7      img_path = os.path.join(img_dir, img_name)
8      img = Image.open(img_path).resize(new_size)
9      img.save(os.path.join(out_dir, img_name))
10
11 def rescale_annotations(image, annotations, new_size):
12     for annotation in annotations:
13         annotation["bbox"][0] *= new_size[0] / image["width"]
14         annotation["bbox"][1] *= new_size[1] / image["height"]
15         annotation["bbox"][2] *= new_size[0] / image["width"]
16         annotation["bbox"][3] *= new_size[1] / image["height"]
17     return annotations
18
19 def write_json(json_dict, out_dir, filename):
20     with open(os.path.join(out_dir, filename), "w") as f:
21         f.write(json.dumps(json_dict))
22
23 def create_empty_manifest(dataset):
24     manifest = {
25         "info": dataset["info"],
26         "licenses": dataset["licenses"],
27         "categories": dataset["categories"],
28         "annotations": [],
29         "images": []
30     }
31     return manifest
32
33 if __name__ == "__main__":
34     # Set COCO dataset paths
35     #dataset_type = "train"
36     dataset_type = "val"
37     data_dir    = "../../Datasets/coco"
38     ann_file    = os.path.join(data_dir, f"annotations/instances_{dataset_type}2014.json")
39     image_dir   = os.path.join(data_dir, f"{dataset_type}2014")
40     output_dir = os.path.join("datasets", dataset_type)
41     os.makedirs(output_dir, exist_ok=True)
42
43     # Load COCO dataset
44     coco = COCO(ann_file)
45
46     # Build up a list of all the images that contain at least one of any category
47     categories = ["pizza", "cat", "bus"]
48     cat_ids = coco.getCatIds(catNms=categories)
49     img_ids = [ coco.getImgIds(catIds=cat_id) for cat_id in cat_ids ]
50     img_ids = set(x for xs in img_ids for x in xs) # unique list of images
51
52     # Filter the list of images by those containing objects above minimum size
53     counter = 0
54     new_size = (256,256)
55     new_dataset = create_empty_manifest(coco.dataset)
56     for img_id in img_ids:
57         ann_ids = coco.getAnnIds(imgIds=img_id, catIds=cat_ids, iscrowd=False)
58         annotations = [ann for ann in coco.loadAnns(ann_ids) if ann["area"] > 40000]
59         if len(annotations):
60             image = coco.loadImgs(img_id)[0]
61             new_dataset["images"] += coco.loadImgs(img_id)
62             new_dataset["annotations"] += rescale_annotations(image, annotations, new_size)
63             save_image(image_dir, output_dir, image["file_name"], new_size)
64             counter += 1
65
66     # Write new annotations file to make later processing easier
67     print(counter)
68     write_json(new_dataset, output_dir, f"instances_{dataset_type}.json")
```

Listing 2: Plotting routine for 3x3 example grid

```
if __name__ == "__main__":
    """Create a 3x3 grid of images with each row showing three images of the same class"""
    import matplotlib.pyplot as plt
    train_dataset = Dataset(dataset_type="train")
    transform = tvt.Compose([
        tvt.Normalize([-1.0, -1.0, -1.0], [2.0, 2.0, 2.0]),
        tvt.ToPILImage()
    ])

    fig = plt.figure()
    fig.set_tight_layout(True)
    counts = {i: 0 for i in range(3)}
    for i in range(len(train_dataset)):
        image, bboxes, labels = train_dataset[i]
        image = np.array(transform(image))
        bbox = bboxes[0]
        label = labels[0]
        if counts[label] < 3:
            [x, y, w, h] = bbox
            image = cv2.rectangle(image, (int(x), int(y)), (int(x+w), int(y+h)), (36,255,12)
                , 2)
            image = cv2.putText(image, train_dataset.label_to_name(label),
                (int(x + 10), int(y + 30)), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (36,255,12), 2)
            counts[label] += 1
            plt.subplot(3,3,sum(counts.values()))
            plt.imshow(image)
        elif sum(counts.values()) > 8:
            plt.suptitle("Grid of Images from Each Class")
            plt.show()
            break
```

## 2 Building Your Deep Network

For the multi-instance object localization task, I designed a network based on Professor Kak's `NetForYolo` and `SkipBlock` architecture as provided in his YOLOLogic library. In my case, I decided to use the following YOLO parameters:

| | |
|---|---|
| Grid Size | $32 \times 32$ pixels |
| Number of Cells | $8 \times 8$ |
| Anchor Boxes | $(5, 3, 1, 1/3, 1/5)$ |
| Number of Classes | 3 |
| Number of Learnable Parameters | $59,177,664$ |

Table 1: Table of `MyNetForYolo` Parameters

This results in a `yolo_tensor` that has the following shape:

$$\begin{aligned} \texttt{yolo\_tensor} &= [\texttt{batch\_size}, \texttt{number\_of\_cells}, \texttt{number\_of\_anchors}, 5 + \texttt{number\_of\_classes}] \\ &= [\texttt{batch\_size}, 64, 5, 8] \end{aligned}$$

Note that I did **not** take Professor Kak's approach of adding a ninth parameter to the `yolo_tensor` to represent the probability that no class is present in a given cell/anchor pair. I found that adding a ninth parameter hampered the performance of my network, as, given the sparsity of objects across all the cells/anchors, the probability mass would concentrate on this ninth parameter and prevent my network for detecting any objects. Also, after reviewing the original YOLO paper, I found that the ninth parameter seemed to contain redundant information, as the first parameter in the `yolo_tensor` is supposed to represent the "confidence" that an object lies in the given cell/anchor pair anyway. Instead, to detect the presence of an object, I thresholded the value of `yolo_tensor[:,:,:,0]` (usually by 0.25, which I determined empirically

to work well). The network implementation is shown in Listing 3. Note: I originally shared this approach with others in this class in Question 245 on Piazza as "Anonymous Comp". Others have since adopted the approach and found success.

Listing 3: Network Architecture for YOLO (`yolo_network.py`)

```
import torch
import torch.nn as nn
import numpy as np
from dataloader_coco import Dataset

class YoloParameters:
    required_keywords = [
        "yolo_interval",
        "num_classes",
        "epochs",
        "batch_size",
        "anchors",
        "image_size",
        "learning_rate",
        "momentum",
        "threshold",
        "display_count"
    ]

    def __init__(self, **kwargs):
        if not all(key in kwargs for key in self.required_keywords):
            raise ValueError(f"Missing a required keyword! Required: {self.required_keywords}")
        for key, value in kwargs.items():
            setattr(self, key, value)
        self.cells = (self.image_size[0] // self.yolo_interval, self.image_size[1] // self.yolo_interval)
        self.num_cells = self.cells[0] * self.cells[1]
        self.anchors = np.asarray(self.anchors)
        self.num_anchors = self.anchors.size
        self.device = torch.device("cuda:0")

class SkipBlock(nn.Module):
    """Adapted from DLStudio.SkipBlock"""
    def __init__(self, in_ch, out_ch, downsample=False, skip_connections=True):
        super().__init__()
        self.downsample = downsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.convo1 = nn.Conv2d(in_ch, in_ch, 3, stride=1, padding=1)
        self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(in_ch)
        self.bn2 = nn.BatchNorm2d(out_ch)
        self.in2out = nn.Conv2d(in_ch, out_ch, 1)
        if downsample:
            ##  Setting stride to 2 and kernel_size to 1 amounts to retaining every
            ##  other pixel in the image --- which halves the size of the image:
            self.downsampler1 = nn.Conv2d(in_ch, in_ch, 1, stride=2)
            self.downsampler2 = nn.Conv2d(out_ch, out_ch, 1, stride=2)

    def forward(self, x):
        identity = x
        out = self.convo1(x)
        out = self.bn1(out)
        out = nn.functional.relu(out)
        out = self.convo2(out)
        out = self.bn2(out)
        out = nn.functional.relu(out)
        if self.downsample:
            identity = self.downsampler1(identity)
            out = self.downsampler2(out)
```

```
61          if self.skip_connections:
62              if (self.in_ch == self.out_ch) and (self.downsample is False):
63                  out = out + identity
64              elif (self.in_ch != self.out_ch) and (self.downsample is False):
65                  identity = self.in2out( identity )
66                  out = out + identity
67              elif (self.in_ch != self.out_ch) and (self.downsample is True):
68                  out = out + torch.cat((identity, identity), dim=1)
69          return out
70
71  class MyNetForYolo(nn.Module):
72      """Adapted from YOLOLogic.YoloObjectDetector.NetForYolo"""
73      def __init__(self, yolo, skip_connections=True, depth=2):
74          super().__init__()
75          self.depth = depth // 2
76          self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
77          self.conv2 = nn.Conv2d(64, 64, 3, padding=1)
78          self.pool = nn.MaxPool2d(2, 2)
79          self.bn1  = nn.BatchNorm2d(64)
80          self.bn2  = nn.BatchNorm2d(128)
81          self.bn3  = nn.BatchNorm2d(256)
82          self.skip64_arr = nn.ModuleList()
83          for i in range(self.depth):
84              self.skip64_arr.append(SkipBlock(64, 64, skip_connections=skip_connections))
85          self.skip64ds = SkipBlock(64,64,downsample=True, skip_connections=skip_connections)
86          self.skip64to128 = SkipBlock(64, 128, skip_connections=skip_connections)
87          self.skip128_arr = nn.ModuleList()
88          for i in range(self.depth):
89              self.skip128_arr.append(SkipBlock(128,128,skip_connections=skip_connections))
90          self.skip128ds = SkipBlock(128,128,downsample=True,skip_connections=skip_connections
                )
91          self.skip128to32 = SkipBlock(128, 32, skip_connections=skip_connections)
92          self.skip256_arr = nn.ModuleList()
93          for i in range(self.depth):
94              self.skip256_arr.append(SkipBlock(256,256,skip_connections=skip_connections))
95          self.skip256ds = SkipBlock(256,256,downsample=True, skip_connections=
                skip_connections)
96          self.fc_seqn = nn.Sequential(
97              nn.Linear(8192, 4096),
98              nn.ReLU(inplace=True),
99              nn.Linear(4096, 2048),
100             nn.ReLU(inplace=True),
101             nn.Linear(2048, yolo.num_cells * yolo.num_anchors * (5+yolo.num_classes))
102         )
103
104     def forward(self, x):
105         x = self.pool(torch.nn.functional.relu(self.conv1(x)))
106         x = nn.MaxPool2d(2,2)(torch.nn.functional.relu(self.conv2(x)))
107         for i,skip64 in enumerate(self.skip64_arr[:self.depth//4]):
108             x = skip64(x)
109         x = self.skip64ds(x)
110         for i,skip64 in enumerate(self.skip64_arr[self.depth//4:]):
111             x = skip64(x)
112         x = self.bn1(x)
113         x = self.skip64to128(x)
114         for i,skip128 in enumerate(self.skip128_arr[:self.depth//4]):
115             x = skip128(x)
116         x = self.bn2(x)
117         x = self.skip128ds(x)
118         x = self.skip128to32(x)
119         x = x.view(-1, 8192 )
120         x = self.fc_seqn(x)
121         return x
```

# 3 Training and Evaluating Your Network

## 3.1 Dataloading Logic

For dataloading, I used the annotations manifest I created when extracting the images to generate a custom Torch dataset for images, labels, and bounding boxes (Listing 4). I then wrapped that dataset in a Torch Dataloader instance to set batch size and shuffling of the data (Listing 5, Lines 175–182). As each batch of the dataset is loaded into the network, I wrote a method called `create_yolo_tensor` to encode the image's bounding box and label into the expected output of the network. Important steps included:

- Assigning each object in the image to the nearest grid cell
- Assigned each object to the anchor box with the closest aspect ratio
- Converting bounding box position to pixels relative to the nearest grid cell (to keep values in $[-1, 1]$)
- Converting bounding box height/width to fractions of the overall image width (to keep values in $[0, 1]$)
- Assigning the object label with one-hot encoding
- Setting the confidence parameter to one with an object is present in a cell/anchor pair

Listing 4: Implementation of Custom Dataset (`dataloader_coco.py`)

```
1  from pycocotools.coco import COCO
2  from PIL import Image
3  import os
4  import sys
5  import cv2
6  import torch
7  import numpy as np
8  import torchvision.transforms as tvt
9  import torch
10
11 class Dataset(torch.utils.data.Dataset):
12     def __init__(self, dataset_type):
13         # Generate list of all image_ids in the dataset_type=train or dataset_type=val
               dataset
14         self.data_dir = os.path.join("datasets", dataset_type)
15         self.coco = COCO(os.path.join(self.data_dir, f"instances_{dataset_type}.json"))
16         self.ids = [img["id"] for img in self.coco.dataset["images"]]
17
18         # Create map of COCO labels to zero-indexed labels for one-hot encoding
19         _, labels = self.load_annotations(self.ids)
20         self.label_map = {label: i for i, label in enumerate(set(labels))}
21         self.label_map_names = {i: self.coco.loadCats(label)[0]["name"] for label, i in self
               .label_map.items()}
22
23         # Define method for converting PIL image format to [-1.0, 1.0] tensor
24         self.transform = tvt.Compose([
25             tvt.ToTensor(),
26             tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
27         ])
28
29     def __len__(self):
30         return len(self.ids)
31
32     def __getitem__(self, index):
33         img_id = self.ids[index]
34         image = self.load_image(img_id)
35         bboxes, labels = self.load_annotations(img_id)
36         mapped_labels = [self.label_map[label] for label in labels]
37         return image, bboxes, mapped_labels
38
39     def load_annotations(self, img_ids):
40         annotations = self.coco.loadAnns(self.coco.getAnnIds(imgIds=img_ids, iscrowd=False))
```

```
        bboxes = [ann["bbox"] for ann in annotations]
        labels = [ann["category_id"] for ann in annotations]
        return bboxes, labels

    def load_image(self, img_id):
        info = self.coco.loadImgs(img_id)[0]
        return self.transform(Image.open(os.path.join(self.data_dir, info["file_name"])).
            convert("RGB"))

    def label_to_name(self, label):
        return self.label_map_names[label]

    def collate(batch):
        images = torch.tensor(np.array([item[0] for item in batch]))
        bboxes = [item[1] for item in batch]
        labels = [item[2] for item in batch]
        return images, bboxes, labels
```

### 3.2 Training Logic

The training logic was fairly routine, except now I had to use multiple loss functions. Thankfully, Torch makes it easy to apply separate loss functions to the appropriate `yolo_tensor` parameters. The losses of each component were summed together to yield the total loss, as shown in Figure 2. To check the accuracy during training, I wrote a short method to convert the raw network outputs for confidence and classification into actual probabilities using Sigmoid and Softmax respectively. The check loop simply prints the ground truth labels/bboxes and predicted labels/bboxes, which allowed me to watch the convergence of the network in real time.

Listing 5: Implementation of Training Logic (`train_network.py`)

```
import torch
import torch.nn as nn
import numpy as np
from dataloader_coco import Dataset
from yolo_network import YoloParameters, MyNetForYolo
import torch.optim as optim
import torchvision
import time
import matplotlib.pyplot as plt

class Logger:
    def __init__(self, filename):
        self.file = open(filename, "w")
        self.history = []

    def write(self, *values):
        self.history.append(values)
        for value in values:
            self.file.write("%.3f," % value)
        self.file.write("\n")
        self.file.flush()

    def __del__(self):
        self.file.close()

def create_yolo_tensor(yolo, bboxes, labels):
    yolo_tensor = torch.zeros(yolo.batch_size, yolo.num_cells, yolo.num_anchors, 5+yolo.
        num_classes)
    yolo_tensor[:,:,:,3:5] = 0.75                    # Make the prior assumption that bbox fills
        most of cell
    yolo_tensor[:,:,:,5:] = 1.0/yolo.num_classes # Force uniform probability mass
    for ibx in range(len(labels)):
        for bbox, label in zip(bboxes[ibx], labels[ibx]):
            # Find the nearest cell and determine the distance between the GT bbox and the
                cell center
```

```
33              bb_center = [
34                  bbox[0] + 0.5*bbox[2],
35                  bbox[1] + 0.5*bbox[3]
36              ]
37              nearest_cell = [
38                  int(np.clip(bb_center[0] // yolo.yolo_interval, 0, yolo.cells[0]-1)),
39                  int(np.clip(bb_center[1] // yolo.yolo_interval, 0, yolo.cells[1]-1))
40              ]
41              nearest_cell_center = [ (a+0.5)*yolo.yolo_interval for a in nearest_cell ]
42              delta = [ (a-b) / yolo.yolo_interval for a, b in zip(bb_center,
                    nearest_cell_center) ]
43              boxdims = [ a/b for a, b in zip(bbox[2:4], yolo.image_size) ]
44
45              # Determine the proper aspect ratio anchor box to use
46              aspect_ratio = bbox[2] / bbox[3]
47              if np.isnan(aspect_ratio):
48                  anchor_idx = len(yolo.anchors)-1
49              else:
50                  anchor_idx = np.argmin( np.abs(aspect_ratio - yolo.anchors) )
51
52              # Encode the GT object information in a vector
53              yolo_vector = torch.zeros(5 + yolo.num_classes)
54              yolo_vector[0] = 1
55              yolo_vector[1] = delta[0]
56              yolo_vector[2] = delta[1]
57              yolo_vector[3] = boxdims[0]
58              yolo_vector[4] = boxdims[1]
59              yolo_vector[5 + label] = 1
60              cell_idx = nearest_cell[1]*yolo.cells[0] + nearest_cell[0]
61              yolo_tensor[ibx, cell_idx, anchor_idx] = yolo_vector
62
63      return yolo_tensor
64
65
66  def run_training_loop(yolo, net, train_dataloader, label_map_func, display_labels=True,
        display_images=False, debug=False):
67      criterion1 = nn.BCELoss(reduction="sum")
68      criterion2 = nn.MSELoss(reduction="sum")
69      criterion3 = nn.CrossEntropyLoss(reduction="sum")
70      logger = Logger("performance_numbers_model3.txt")
71      net = net.to(yolo.device)
72
73      number_of_learnable_params = sum(p.numel() for p in net.parameters() if p.requires_grad)
74      print(number_of_learnable_params)
75
76      optimizer = optim.Adam(net.parameters(), lr=yolo.learning_rate)
77      start_time = time.perf_counter()
78      elapsed_time = 0.0
79
80      print("\n\nStarting training loop...\n\n")
81      for epoch in range(yolo.epochs):
82          running_loss = 0.0
83          running_bceloss = 0.0
84          running_regloss = 0.0
85          running_labloss = 0.0
86          for i, (images, bboxes, labels) in enumerate(train_dataloader):
87              yolo_tensor = create_yolo_tensor(yolo, bboxes, labels).to(yolo.device)
88              images = images.to(yolo.device)
89
90              optimizer.zero_grad()
91              output = net(images)
92              predictions = output.view(-1, yolo_tensor.shape[1], yolo_tensor.shape[2],
                    yolo_tensor.shape[3])
93
94              loss = torch.tensor(0.0, requires_grad=True).float().to(yolo.device)
95
96              ##  Estimating presence/absence of object with the Binary Cross Entropy loss:
97              bceloss = criterion1(nn.Sigmoid()(predictions[:,:,:,0]), yolo_tensor[:,:,:,0])
```

```
98              loss += bceloss
99
100             ## MSE loss for the regression params for the object bounding boxes:
101             regression_loss = criterion2(predictions[:,:,:,1:5], yolo_tensor[:,:,:,1:5])
102             loss += regression_loss
103
104             ##  CrossEntropy loss for object class labels:
105             targets = yolo_tensor[:,:,:,5:]
106             targets = targets.view(-1, yolo.num_classes)
107             targets = torch.argmax(targets, dim=1)
108             probs = predictions[:,:,:,5:]
109             probs = probs.view(-1, yolo.num_classes)
110             class_labeling_loss = criterion3(probs, targets)
111             loss += class_labeling_loss
112
113             ## Backpropagation
114             loss.backward()
115             optimizer.step()
116
117             ## Update all running losses
118             running_loss += loss.item()
119             running_bceloss += bceloss.item()
120             running_regloss += regression_loss.item()
121             running_labloss += class_labeling_loss.item()
122
123             with torch.no_grad():
124                 if i % yolo.display_count == yolo.display_count-1:
125                     current_time = time.perf_counter()
126                     elapsed_time = current_time - start_time
127                     print("\n[epoch:%d/%d, iter=%4d  elapsed_time=%5d secs]" % (epoch+1,yolo
                            .epochs, i+1, elapsed_time))
128
129                     print(running_loss, running_bceloss, running_regloss, running_labloss)
130                     logger.write(
131                         running_loss / yolo.display_count,
132                         running_bceloss / yolo.display_count,
133                         running_regloss / yolo.display_count,
134                         running_labloss / yolo.display_count
135                     )
136                     running_loss = 0.0
137                     running_bceloss = 0.0
138                     running_regloss = 0.0
139                     running_labloss = 0.0
140
141                     # Convert raw output to probabilities
142                     predictions[:,:,:,0] = nn.Sigmoid()(predictions[:,:,:,0])
143                     predictions[:,:,:,5:] = nn.Softmax(dim=-1)(predictions[:,:,:,5:])
144
145                     # Show the detected labels
146                     for ibx in range(predictions.shape[0]):
147                         detection_mask = predictions[ibx,:,:,0] > yolo.threshold
148                         detections = predictions[ibx, detection_mask]
149                         detected_labels = [
150                             label_map_func(label.item()) for label in torch.argmax(
                                    detections[:,5:], dim=-1)
151                         ]
152                         truth_mask = yolo_tensor[ibx,:,:,0] == 1
153                         truth = yolo_tensor[ibx, truth_mask]
154                         truth_labels = [
155                             label_map_func(label.item()) for label in torch.argmax(truth
                                    [:,5:], dim=-1)
156                         ]
157                         print(f"Batch: {ibx}")
158                         print(f"Truth: {truth_labels} {truth[:,1:5]}")
159                         print(f"Detected: {detected_labels} {detections[:,1:5]}\n")
160
161     torch.save(net.state_dict(), "saved_model3")
162
```

```
if __name__ == "__main__":
    yolo = YoloParameters(
        yolo_interval=32,
        num_classes=3,
        epochs=33,
        batch_size=8,
        anchors=(5.0, 3.0, 1.0, 1.0/3.0, 1.0/5.0),
        image_size=(256,256),
        learning_rate=1e-4,
        momentum=0.9,
        threshold=0.25,
        display_count=100
    )
    train_dataset = Dataset(dataset_type="train")
    train_dataloader = torch.utils.data.DataLoader(
        train_dataset,
        batch_size=yolo.batch_size,
        shuffle=True,
        num_workers=2,
        collate_fn=Dataset.collate
    )

    net = MyNetForYolo(yolo, depth=12)
    run_training_loop(yolo, net, train_dataloader, label_map_func=train_dataset.
        label_to_name)
```

### 3.3 Testing Logic

The testing logic was slightly more involved. For each image and corresponding network output, I had to first determine if the network had actually detected any objects in the image. I did this using the threshold approach I described in the previous section. In my case, if the confidence parameter of a given cell/anchor pair exceeded 0.25, I declared an object present. Otherwise, I declared no object present. Next, I had to again convert the Yolo-encoded bounding boxes into image bounding boxes using the image grid and size parameters. Finally, I had to plot the ground-truth and detected image labels and bounding boxes on the image before displaying with `matplotlib`.

As an additional measure, I wrote a short routine to determine if for a given image the network correctly or incorrectly detected all the objects in the image. I used this routine to assemble a "good" list and a "bad" list for each of the classes, thereby generating the image grids in Figures 3 and 4.

Listing 6: Implementation of Testing Logic (`test_network.py`)

```
import torch
import torch.nn as nn
import numpy as np
from dataloader_coco import Dataset
from yolo_network import YoloParameters, MyNetForYolo
import torchvision
import torchvision.transforms as tvt
import time
import cv2
import matplotlib.pyplot as plt
from train_network import create_yolo_tensor

def filter_bboxes_and_labels(yolo_tensor, threshold, label_map_func):
    truth_mask = yolo_tensor[:,:,0] >= threshold
    truth = yolo_tensor[truth_mask]
    if truth.shape[0] > 0:
        bboxes = truth[:,1:5].tolist()
        labels = [ label_map_func(label.item()) for label in torch.argmax(truth[:,5:], dim
            =-1) ]
        cells  = torch.arange(truth_mask.shape[0])[torch.any(truth_mask, dim=-1)].tolist()
        return bboxes, labels, cells
    return [], [], []

```

```
23  def convert_yolo_to_image_bbox(yolo, yolo_bbox, yolo_cell):
24      cell_idxs = [
25          yolo_cell % yolo.cells[0], # x
26          yolo_cell // yolo.cells[1] # y
27      ]
28      xc = (cell_idxs[0] + yolo_bbox[0]) * yolo.yolo_interval
29      yc = (cell_idxs[1] + yolo_bbox[1]) * yolo.yolo_interval
30      w = np.clip(yolo_bbox[2] * yolo.image_size[0], 0, yolo.image_size[0])
31      h = np.clip(yolo_bbox[3] * yolo.image_size[1], 0, yolo.image_size[1])
32      x = np.clip(xc - 0.5*w, 0, yolo.image_size[0])
33      y = np.clip(yc - 0.5*h, 0, yolo.image_size[1])
34      return x, y, w, h
35
36  def add_yolo_bboxes_to_image(yolo, image, yolo_bboxes, labels, cells, color=(36,255,12)):
37      for bbox, label, cell in zip(yolo_bboxes, labels, cells):
38          x, y, w, h = convert_yolo_to_image_bbox(yolo, bbox, cell)
39          image = cv2.rectangle(image, (int(x), int(y)), (int(x+w), int(y+h)), color, 2)
40          image = cv2.putText(image, label, (int(x+10), int(y+20)), cv2.FONT_HERSHEY_SIMPLEX,
                0.8, color, 2)
41      return image
42
43  def run_testing_loop(yolo, net, test_dataloader, label_map_func):
44      tensor_to_image = tvt.Compose([
45          tvt.Normalize([-1.0, -1.0, -1.0], [2.0, 2.0, 2.0]),
46          tvt.ToPILImage()
47      ])
48
49      good_image_map = {a: [] for a in ["cat", "bus", "pizza"]}
50      good_images_per_class = 6
51      good_image_counter = 0
52      bad_image_map = {a: [] for a in ["cat", "bus", "pizza"]}
53      bad_images_per_class = 2
54      bad_image_counter = 0
55      for i, (images, bboxes, labels) in enumerate(test_dataloader):
56          yolo_tensor = create_yolo_tensor(yolo, bboxes, labels)
57          images = images.to(yolo.device)
58          with torch.no_grad():
59              output = net(images)
60              predictions = output.view(-1, yolo_tensor.shape[1], yolo_tensor.shape[2],
                    yolo_tensor.shape[3])
61              predictions[:,:,:,0] = nn.Sigmoid()(predictions[:,:,:,0])
62              predictions[:,:,:,5:] = nn.Softmax(dim=-1)(predictions[:,:,:,5:])
63              predictions = predictions.to("cpu")
64
65              for ibx in range(images.shape[0]):
66                  truth_bboxes, truth_labels, truth_cells = filter_bboxes_and_labels(
                        yolo_tensor[ibx], 1.0, label_map_func)
67                  pred_bboxes, pred_labels, pred_cells = filter_bboxes_and_labels(predictions[
                        ibx], yolo.threshold, label_map_func)
68                  this_image = np.array(tensor_to_image(images[ibx].to("cpu")))
69                  this_image = add_yolo_bboxes_to_image(yolo, this_image, truth_bboxes,
                        truth_labels, truth_cells)
70                  this_image = add_yolo_bboxes_to_image(yolo, this_image, pred_bboxes,
                        pred_labels, pred_cells, color=(255,0,0))
71
72                  # If we found the right classes, add this_image to the good list
73                  if all(plabel in truth_labels for plabel in pred_labels) and len(pred_labels
                        ) == len(truth_labels):
74                      not_added = True
75                      good_image_counter += 1
76                      for plabel in pred_labels:
77                          if len(good_image_map[plabel]) < good_images_per_class and not_added
                                :
78                              good_image_map[plabel].append(this_image)
79                              not_added = False
80
81                  # If we found the wrong classes (or none at all), add this_image to the bad
                        list
```

```
                #if not any(plabel in truth_labels for plabel in pred_labels):
                else:
                    not_added = True
                    bad_image_counter += 1
                    for tlabel in truth_labels:
                        if len(bad_image_map[tlabel]) < bad_images_per_class and not_added:
                            bad_image_map[tlabel].append(this_image)
                            not_added = False

    print(good_image_counter, bad_image_counter, good_image_counter + bad_image_counter)

    if all(len(a) >= 0 for a in good_image_map.values()):
        fig = plt.figure()
        fig.set_tight_layout(True)
        for row, image_set in enumerate(good_image_map.values()):
            for col, image in enumerate(image_set):
                plt.subplot(len(good_image_map), good_images_per_class, row*
                    good_images_per_class + col + 1)
                plt.imshow(image)
        plt.suptitle("Grid of Successfully Identified Images")
        plt.show()

    if all(len(a) >= 0 for a in bad_image_map.values()):
        fig = plt.figure()
        fig.set_tight_layout(True)
        for row, image_set in enumerate(bad_image_map.values()):
            for col, image in enumerate(image_set):
                plt.subplot(len(bad_image_map), bad_images_per_class, row*
                    bad_images_per_class + col + 1)
                plt.imshow(image)
        plt.suptitle("Grid of Incorrectly Identified Images")
        plt.show()


if __name__ == "__main__":
    yolo = YoloParameters(
        yolo_interval=32,
        num_classes=3,
        epochs=33,
        batch_size=8,
        anchors=(5.0, 3.0, 1.0, 1.0/3.0, 1.0/5.0),
        image_size=(256,256),
        learning_rate=1e-4,
        momentum=0.9,
        threshold=0.25,
        display_count=100
    )
    test_dataset = Dataset(dataset_type="val")
    test_dataloader = torch.utils.data.DataLoader(
        test_dataset,
        batch_size=yolo.batch_size,
        shuffle=True,
        num_workers=2,
        collate_fn=Dataset.collate
    )

    net = MyNetForYolo(yolo, depth=12)
    net.load_state_dict(torch.load("saved_model3", weights_only=True))
    net = net.to(yolo.device)
    net.eval()

    run_testing_loop(yolo, net, test_dataloader, label_map_func=test_dataset.label_to_name)
```

### 3.4 Performance

On the whole, the network is mostly functional as multi-object detector and localizer. I had an absolutely terrible time getting this network to converge. I think a large part of my issues were with the encoding of the `yolo_tensor`. Through proper scaling of parameters and dropping the extra probability mass term, I was able to improve my performance enough to converge to the model I have now, which is able to detect and localize with moderate success, as shown in the image grids below. When the model detects the presence of an object, it generally does well classifying it and determining an accurate bounding box. More often than not, though, the network struggles to detect the presence of any objects with the thresholding approach. At a threshold of 0.25, the network corrected the correct object (and proper number of objects) approximately 40% of the time. I think the issue is not that the network failed to detect, so much as my thresholding and bounding box selection method is lacking. My probability of false alarm is low, but my probability of detection is lower than I would like with this approach. With better investigation of the threshold term or some of the fancy multiple bounding box decisionmaking discussed on Piazza, I bet I could improve that score a lot.

I also suspect that I might be able to improve performance further by:

- Encoding all `yolo_tensor` parameters to be between 0 and 1. Scaling the bounding box parameters by cell size instead of image size allowed the height and width to exceed the range of $[-1, 1]$ and significantly reduced the performance of my network, as far as I could tell.
- Further investigating network layout and parameter counts. I would specifically like to implement the original YOLO network and test its performance on the same dataset.
- Increasing the number of training images, either through real data or augmentation.
- Investigating regularization as a method to avoid overfitting, as judging from the training loss vs testing loss performance, I definitely overtrained some of my earlier models.

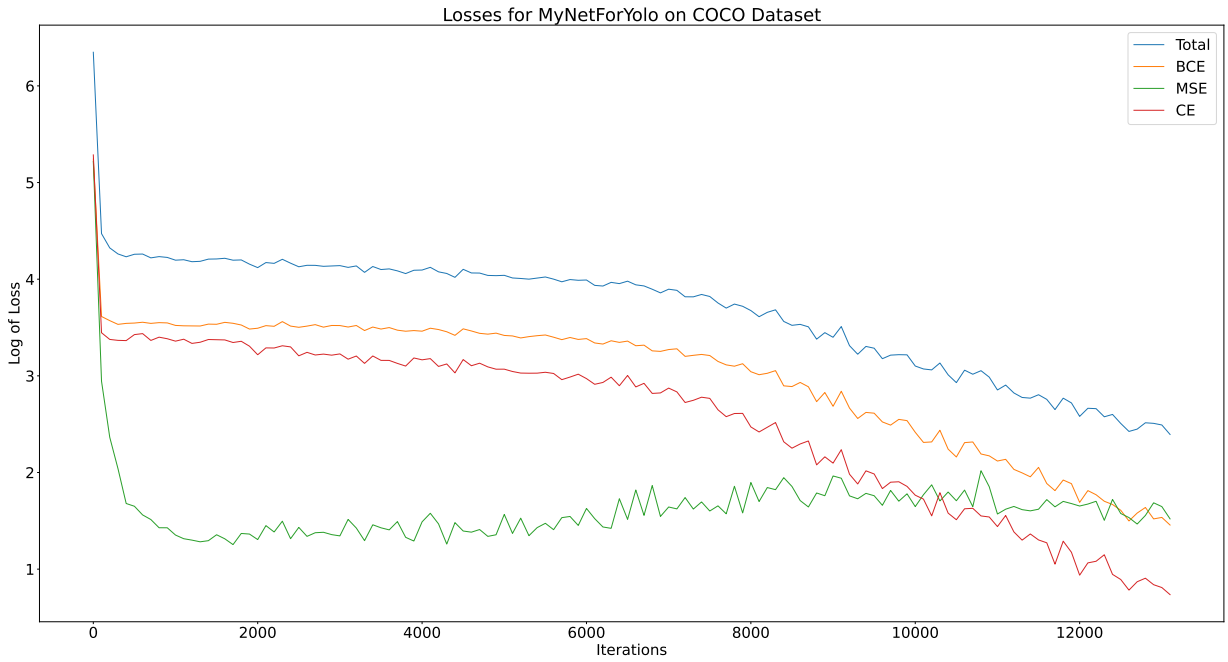


Figure 2: Training loss for each of the three loss functions in `MyNetForYolo`, where BCE is the binary cross-entropy loss for the confidence parameter, MSE is the mean-square error loss for bounding boxes, and CE is the cross-entropy loss for image classifier.

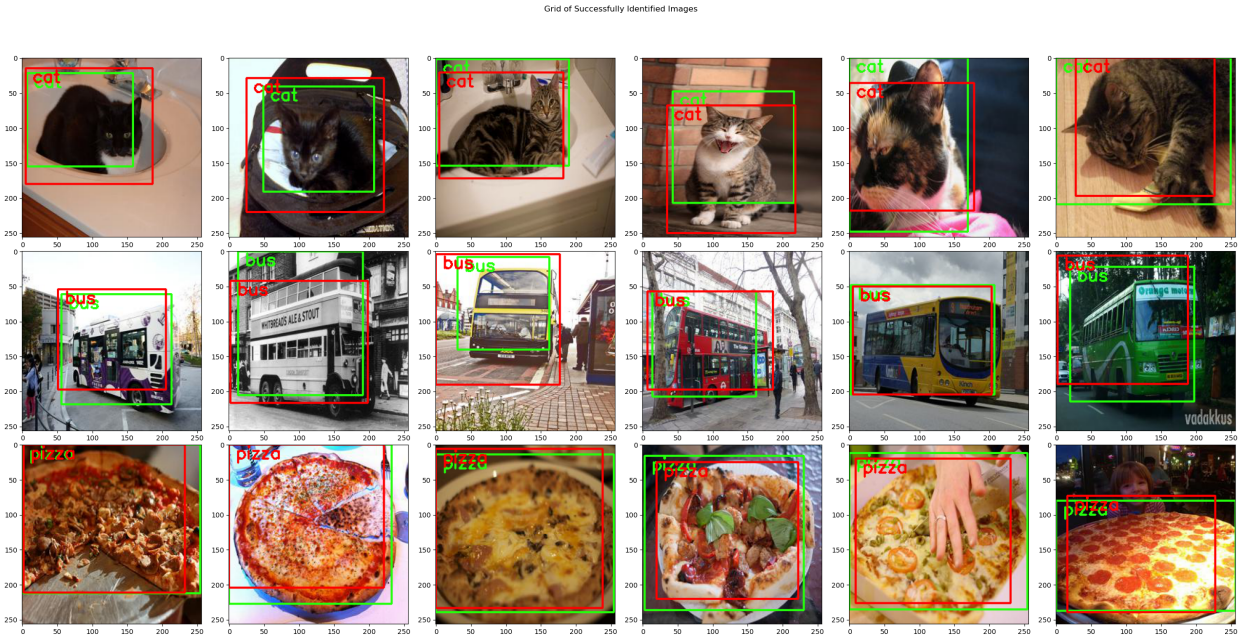


Figure 3: Examples of correctly identified images with ground-truth labels/bboxes in green and predicted labels/bboxes in red for each of the three classes: cat (row 1), bus (row 2), and pizza (row 3).

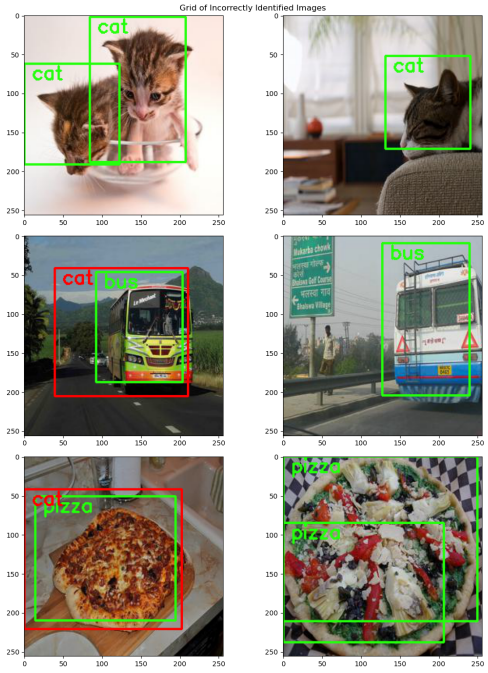


Figure 4: Examples of incorrectly identified images with ground-truth labels/bboxes in green and predicted labels/bboxes in red for each of the three classes: cat (row 1), bus (row 2), and pizza (row 3). For images where no red box appears, my network failed to detect the presence of any object in the image.