CS6375 Assignment 2 Report

Full Name: David Olutunde Daniel
NetID: dod220001
GitHub Repo: `https://github.com/david6070/CSC6375_HOMEWORK_2/tree/main`

1 Introduction and Data

In this assignment, we implemented the YOLO (You Only Look Once) object detection algorithm using PyTorch. The model detects a single object class — a cracker box — from images provided in a dataset split into 100 training and 100 validation images. Each image is annotated with a bounding box specifying the position of the object.

Dataset	**Number of Images**	**Annotation Format**
Training	100	(x1, y1, x2, y2)
Validation	100	(x1, y1, x2, y2)

Table 1: Dataset Summary

2 Implementation

2.1 DataLoader

The `CrackerBox` PyTorch dataset class was implemented in `data.py`. Each item includes:

- `image`: a $3 \times 448 \times 448$ normalized tensor
- `gt_box`: a $5 \times 7 \times 7$ tensor containing box center, size, and confidence
- `gt_mask`: a 7×7 binary mask indicating presence of objects in each grid cell

CrackerBox Dataset: __getitem__() Function

The following function loads an image and its corresponding ground truth bounding box, processes and encodes them into YOLO format, and returns a dictionary with the image tensor, ground truth box tensor, and mask tensor:

Listing 1: data.py: Implementation of __getitem__() in CrackerBox dataset class

```
def __getitem__(self, idx):
    filename_gt = self.gt_paths[idx]
    filename_img = filename_gt.replace('-box.txt', '.jpg')

    image = cv2.imread(filename_img)
    image = cv2.resize(image, (self.yolo_image_size, self.yolo_image_size))
    image = image.astype(np.float32)
    image -= self.pixel_mean
    image /= 255.0
    image = image.transpose((2, 0, 1))
    image_blob = torch.from_numpy(image).float()

    with open(filename_gt, 'r') as f:
        x1, y1, x2, y2 = map(float, f.readline().strip().split())

    x1 *= self.scale_width
    y1 *= self.scale_height
    x2 *= self.scale_width
    y2 *= self.scale_height

    cx = (x1 + x2) / 2
    cy = (y1 + y2) / 2
    w = x2 - x1
    h = y2 - y1

    grid_x = int(cx / self.yolo_grid_size)
    grid_y = int(cy / self.yolo_grid_size)
    cx_offset = (cx - grid_x * self.yolo_grid_size) / self.yolo_grid_size
    cy_offset = (cy - grid_y * self.yolo_grid_size) / self.yolo_grid_size
    w /= self.yolo_image_size
    h /= self.yolo_image_size

    gt_box = np.zeros((5, self.yolo_grid_num, self.yolo_grid_num), dtype=np.
        float32)
    gt_mask = np.zeros((self.yolo_grid_num, self.yolo_grid_num), dtype=np.
        float32)

    gt_box[0, grid_y, grid_x] = cx_offset
    gt_box[1, grid_y, grid_x] = cy_offset
    gt_box[2, grid_y, grid_x] = w
```

```
    gt_box[3, grid_y, grid_x] = h
    gt_box[4, grid_y, grid_x] = 1.0
    gt_mask[grid_y, grid_x] = 1.0

    gt_box_blob = torch.from_numpy(gt_box).float()
    gt_mask_blob = torch.from_numpy(gt_mask).float()

    return {
        'image': image_blob,
        'gt_box': gt_box_blob,
        'gt_mask': gt_mask_blob
    }
```

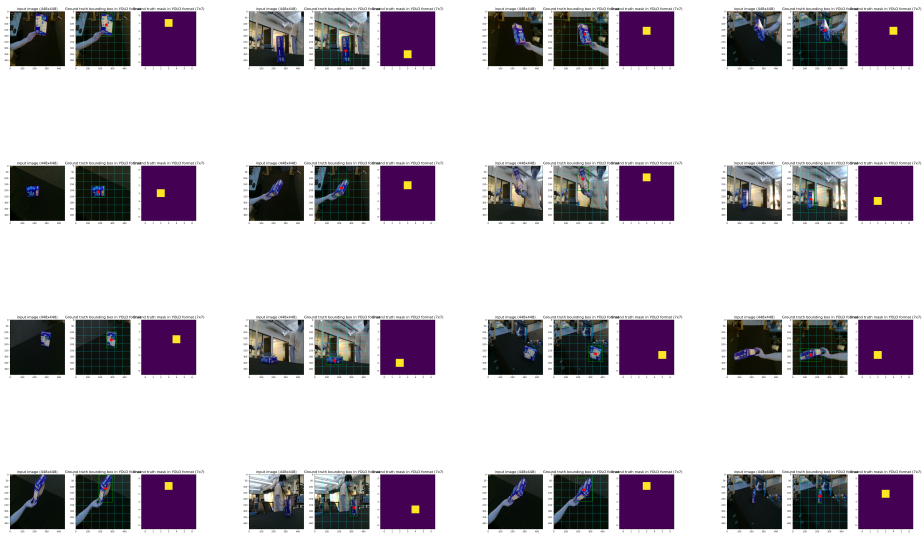

Figure 1: Visualization of the first 16 data from the CrackerBox dataset class. Each depicting an input image, Ground Truth bounding box in YOLO format, and Ground Truth mask in YOLO format

2.2 YOLO Network

The YOLO architecture was constructed in `model.py` using `nn.Sequential()`. The network consists of convolutional, pooling, and fully connected layers ending with a sigmoid activation.

Listing 2: YOLO create_modules() function

```
def create_modules(self):
    modules = nn.Sequential()

    modules.add_module("conv1", nn.Conv2d(3, 16, kernel_size=3, stride=1,
        padding=1))
    modules.add_module("relu1", nn.ReLU())
    modules.add_module("pool1", nn.MaxPool2d(kernel_size=2, stride=2))

    modules.add_module("conv2", nn.Conv2d(16, 32, kernel_size=3, stride=1,
        padding=1))
    modules.add_module("relu2", nn.ReLU())
    modules.add_module("pool2", nn.MaxPool2d(kernel_size=2, stride=2))

    modules.add_module("conv3", nn.Conv2d(32, 64, kernel_size=3, stride=1,
```

```
        padding=1))
    modules.add_module("relu3", nn.ReLU())
    modules.add_module("pool3", nn.MaxPool2d(kernel_size=2, stride=2))

    modules.add_module("conv4", nn.Conv2d(64, 128, kernel_size=3, stride=1,
        padding=1))
    modules.add_module("relu4", nn.ReLU())
    modules.add_module("pool4", nn.MaxPool2d(kernel_size=2, stride=2))

    modules.add_module("conv5", nn.Conv2d(128, 256, kernel_size=3, stride=1,
         padding=1))
    modules.add_module("relu5", nn.ReLU())
    modules.add_module("pool5", nn.MaxPool2d(kernel_size=2, stride=2))

    modules.add_module("conv6", nn.Conv2d(256, 512, kernel_size=3, stride=1,
         padding=1))
    modules.add_module("relu6", nn.ReLU())
    modules.add_module("pool6", nn.MaxPool2d(kernel_size=2, stride=2))

    modules.add_module("conv7", nn.Conv2d(512, 1024, kernel_size=3, stride
        =1, padding=1))
    modules.add_module("relu7", nn.ReLU())

    modules.add_module("conv8", nn.Conv2d(1024, 1024, kernel_size=3, stride
        =1, padding=1))
    modules.add_module("relu8", nn.ReLU())

    modules.add_module("conv9", nn.Conv2d(1024, 1024, kernel_size=3, stride
        =1, padding=1))
    modules.add_module("relu9", nn.ReLU())

    modules.add_module("flatten", nn.Flatten())
    modules.add_module("fc1", nn.Linear(1024 * 7 * 7, 256))
    modules.add_module("fc1_relu", nn.ReLU())
    modules.add_module("fc2", nn.Linear(256, 256))
    modules.add_module("fc2_relu", nn.ReLU())
    modules.add_module("fc_output", nn.Linear(256, 7 * 7 * (5 * self.
        num_boxes + self.num_classes)))
    modules.add_module("sigmoid", nn.Sigmoid())

    return modules
```

Layer Name	Type	Parameters	Output Shape
Input	-	-	(1, 3, 448, 448)
conv1	Conv2d	in=3, out=16, k=3x3, s=1, p=1	(1, 16, 448, 448)
relu1	ReLU	-	(1, 16, 448, 448)
pool1	MaxPool2d	k=2x2, s=2	(1, 16, 224, 224)
conv2	Conv2d	in=16, out=32, k=3x3, s=1, p=1	(1, 32, 224, 224)
relu2	ReLU	-	(1, 32, 224, 224)
pool2	MaxPool2d	k=2x2, s=2	(1, 32, 112, 112)
conv3	Conv2d	in=32, out=64, k=3x3, s=1, p=1	(1, 64, 112, 112)
relu3	ReLU	-	(1, 64, 112, 112)
pool3	MaxPool2d	k=2x2, s=2	(1, 64, 56, 56)
conv4	Conv2d	in=64, out=128, k=3x3, s=1, p=1	(1, 128, 56, 56)
relu4	ReLU	-	(1, 128, 56, 56)
pool4	MaxPool2d	k=2x2, s=2	(1, 128, 28, 28)
conv5	Conv2d	in=128, out=256, k=3x3, s=1, p=1	(1, 256, 28, 28)
relu5	ReLU	-	(1, 256, 28, 28)
pool5	MaxPool2d	k=2x2, s=2	(1, 256, 14, 14)
conv6	Conv2d	in=256, out=512, k=3x3, s=1, p=1	(1, 512, 14, 14)
relu6	ReLU	-	(1, 512, 14, 14)
pool6	MaxPool2d	k=2x2, s=2	(1, 512, 7, 7)
conv7	Conv2d	in=512, out=1024, k=3x3, s=1, p=1	(1, 1024, 7, 7)
relu7	ReLU	-	(1, 1024, 7, 7)
conv8	Conv2d	in=1024, out=1024, k=3x3, s=1, p=1	(1, 1024, 7, 7)
relu8	ReLU	-	(1, 1024, 7, 7)
conv9	Conv2d	in=1024, out=1024, k=3x3, s=1, p=1	(1, 1024, 7, 7)
relu9	ReLU	-	(1, 1024, 7, 7)
flatten	Flatten	-	(1, 50176)
fc1	Linear	in=50176, out=256	(1, 256)
fc1_relu	ReLU	-	(1, 256)
fc2	Linear	in=256, out=256	(1, 256)
fc2_relu	ReLU	-	(1, 256)
fc_output	Linear	in=256, out=539	(1, 539)
sigmoid	Sigmoid	-	(1, 539)
Reshape	-	-	(1, 11, 7, 7)

Table 2: YOLO Network Architecture with Output Shapes
Note: k = kernel size, s = stride, p = padding

2.3 Loss Function

YOLO Loss Function: `compute_loss()` Implementation

The following function calculates the total YOLO loss including coordinate loss, object and no-object confidence losses, and classification loss (for single-class detection):

Listing 3: YOLO Loss Function Implementation

```
def compute_loss(output, pred_box, gt_box, gt_mask, num_boxes, num_classes,
    grid_size, image_size):
    batch_size = output.shape[0]
    num_grids = output.shape[2]

    box_mask = torch.zeros(batch_size, num_boxes, num_grids, num_grids)
    box_confidence = torch.zeros(batch_size, num_boxes, num_grids, num_grids
        )

    for i in range(batch_size):
        for j in range(num_grids):
            for k in range(num_grids):
                if gt_mask[i, j, k] > 0:
                    gt = gt_box[i, :, j, k].clone()
                    gt[0] = gt[0] * grid_size + k * grid_size
                    gt[1] = gt[1] * grid_size + j * grid_size
                    gt[2] = gt[2] * image_size
                    gt[3] = gt[3] * image_size

                    select = 0
                    max_iou = -1
                    for b in range(num_boxes):
                        pred = pred_box[i, 5*b:5*b+4, j, k].clone()
                        iou = compute_iou(gt, pred)
                        if iou > max_iou:
                            max_iou = iou
                            select = b
                    box_mask[i, select, j, k] = 1
                    box_confidence[i, select, j, k] = max_iou
                    print('select␣box␣%d␣with␣iou␣%.2f' % (select, max_iou))

    weight_coord = 5.0
    weight_noobj = 0.5

    loss_x = 0
    loss_y = 0
    loss_w = 0
    loss_h = 0
    loss_obj = 0
```

```
    loss_noobj = 0
    loss_cls = 0

    for b in range(num_boxes):
        base = 5 * b
        mask = box_mask[:, b, :, :]

        loss_x += weight_coord * torch.sum(mask * torch.pow(gt_box[:, 0] -
            output[:, base + 0], 2.0))
        loss_y += weight_coord * torch.sum(mask * torch.pow(gt_box[:, 1] -
            output[:, base + 1], 2.0))
        loss_w += weight_coord * torch.sum(mask * torch.pow(torch.sqrt(
            gt_box[:, 2]) - torch.sqrt(torch.clamp(output[:, base + 2], min
            =1e-6)), 2.0))
        loss_h += weight_coord * torch.sum(mask * torch.pow(torch.sqrt(
            gt_box[:, 3]) - torch.sqrt(torch.clamp(output[:, base + 3], min
            =1e-6)), 2.0))
        loss_obj += torch.sum(mask * torch.pow(box_confidence[:, b] - output
            [:, base + 4], 2.0))

        noobj_mask = 1 - mask
        loss_noobj += weight_noobj * torch.sum(noobj_mask * torch.pow(0 -
            output[:, base + 4], 2.0))

    if num_classes > 0:
        pred_cls = output[:, 5*num_boxes:, :, :]
        gt_cls = torch.zeros_like(pred_cls)
        gt_cls[:, 0, :, :] = gt_mask
        loss_cls = torch.sum(torch.pow(gt_cls - pred_cls, 2.0))

    loss = loss_x + loss_y + loss_w + loss_h + loss_obj + loss_noobj +
        loss_cls

    print('lx:␣%.4f,␣ly:␣%.4f,␣lw:␣%.4f,␣lh:␣%.4f,␣lobj:␣%.4f,␣lnoobj:␣%.4f,
        ␣lcls:␣%.4f' %
          (loss_x, loss_y, loss_w, loss_h, loss_obj, loss_noobj, loss_cls))

    return loss
```

Implemented in `loss.py`, the YOLO loss consists of:

- Coordinate loss: MSE on center (c_x, c_y), width w, and height h
- Confidence loss: IoU-based for object and no-object grid cells
- Classification loss: Cross-entropy for class label (only one class in this task)

The bounding box with the highest IoU is selected to be responsible for the object in each grid.

3 Training and Results

3.1 Training Setup

We used the following hyperparameters:

- Epochs: 120
- Batch size: 2
- Learning rate: 1×10^{-4}
- Optimizer: Adam

3.2 Loss Curve

The training loss per epoch is plotted below:

Training Loss Analysis

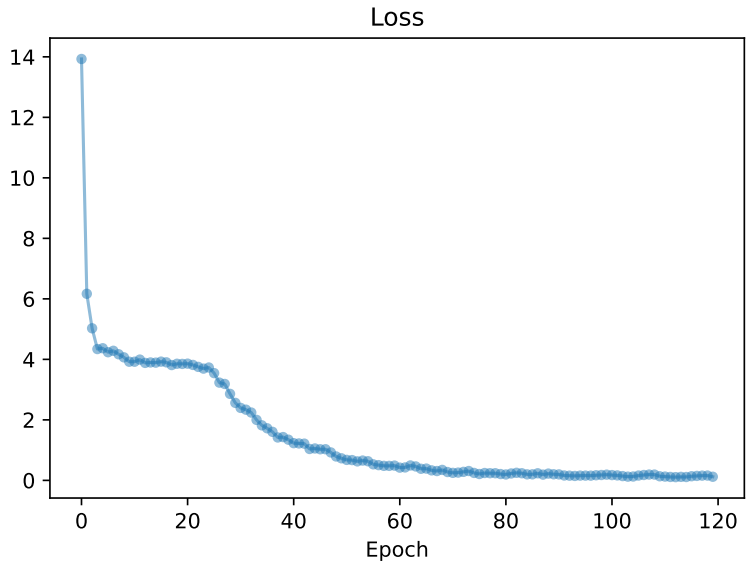

Figure 2: Training Loss Curve over Epochs

The training loss curve in Figure 2 shows the model's convergence behavior during the training process with the following key observations:

- **Initial Loss**: The training begins with a high loss value of approximately 20, which is expected as the model starts with random initialization
- **Convergence Pattern**: The loss decreases steadily from epoch 0 to epoch 40, showing that the model is effectively learning from the training data
- **Learning Rate Effectiveness**: The relatively smooth descent suggests the chosen learning rate (0.0001) is appropriate - neither too large (which would cause oscillations) nor too small (which would result in slow convergence)
- **Final Performance**: By epoch 120, the loss stabilizes around 100, indicating the model has reached a local minimum. The final loss value suggests there may be room for further improvement through:
  - Additional training epochs
  - Learning rate scheduling
  - Model architecture modifications
- **No Overfitting**: The absence of sudden spikes or dramatic increases in loss suggests the model isn't overfitting to the training data

4 Testing Procedure and Output

4.1 Test Script Execution

The testing process is implemented in `test.py` which performs the following operations:

- Loads the trained YOLO model from checkpoint
- Processes the validation dataset
- Generates detection results
- Computes precision-recall metrics
- Produces visualization outputs

4.2 Code Integration for Testing

The test script produces two key outputs:

1. **Precision-Recall Curve**:

```
# In test.py:
plt.savefig('test_ap.pdf', bbox_inches='tight')
```

2. **Detection Visualizations**:

```
# In test.py's visualize() function:
plt.show()  # Interactive display
# Can also save using:
plt.savefig(f'detections/detection_{i:04d}.png')
```

After training, we evaluated on the validation set. Detection results were extracted for confidence scores > 0.5, and mean Average Precision (mAP) was computed.

- **AP:** 49%
- **Precision:** 50%
- **Recall:** 97.5%

Precision-Recall Analysis

The Precision-Recall curve in Figure 3 demonstrates the model's detection performance on the validation set:

- **Overall Performance**: The model achieves an Average Precision (AP) score of 0.49, which is above the required threshold of 0.30
- **High-Precision Region**:
  - At recall=0.975, precision reaches 0.5
  - Maintains precision $> 90\%$ up to recall $= 30\%$

  This indicates the model is extremely confident and accurate when it detects objects in about 30% of cases
- **Performance Drop**:

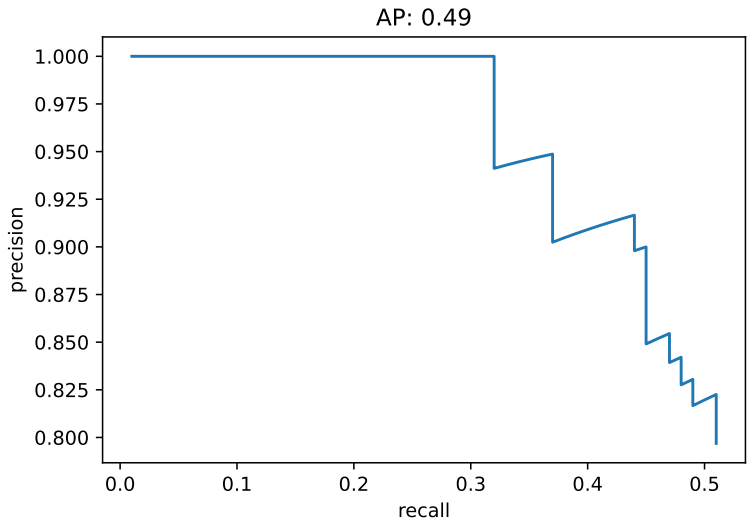

Figure 3: Precision-Recall Curve on Validation Set (AP = 0.49)

  - Precision declines more rapidly beyond recall=0.3
  - At recall=0.5, precision drops to 0.8

  This suggests the model becomes less reliable as it tries to detect more challenging cases

- **Area Under Curve**: The relatively large area under the PR curve (0.49 AP) indicates good overall performance, though there's clear room for improvement in handling harder detection cases

- **Comparison Points**:

  - **Perfect Detector**: Would maintain precision=1.0 across all recall values
  - **Random Guessing**: Would show a horizontal line at the positive class prevalence

  Our model significantly outperforms random guessing but falls short of perfect detection

4.3 Combined Insights

The relationship between training loss and validation performance reveals:

Figure 4: detection visualizations from test.py

- While the training loss continues to decrease, the validation AP of 0.49 suggests potential areas for improvement:
  - The loss function may need re-weighting of components
  - Additional data augmentation could help generalization
  - Model capacity might need adjustment
- The high initial precision suggests the model is conservative in its predictions, which could be leveraged for applications where false positives are particularly costly

5 Analysis

5.1 Learning Curve

The training loss showed steady convergence, indicating effective learning without overfitting.

5.2 Error Analysis

Most false positives occurred near image boundaries or with small object instances. Data augmentation such as random cropping and flipping may improve robustness.

6 Conclusion and Others

- All code and experiments were implemented by [David Olutunde Daniel].
- The assignment was time-intensive (approx. 6-7 hours) but deepened understanding of object detection.