Machine Learning – Assignment 2

Name: Aditya Swaroop

NET ID: axs230571

GitHub Link: https://github.com/as567-code/CS6375_Assignment_2

Introduction and Data

This assignment is all about implementing YOLO (You Only Look Once) object detector using PyTorch. YOLO is pretty cool because it does object detection in a single pass through the network, making it super fast compared to other methods. The paper by Redmon et al. from 2016 introduced this approach, and I'm implementing a version of it to detect cracker boxes.

For this assignment, I worked with a dataset of cracker box images that was split like this:

Dataset	Number of images
Training Set	100
Validation Set	100

The ground truth annotations came in text files with the format (x1, y1, x2, y2) which are the coordinates for the bounding boxes.

Question 1: DataLoader Implementation

For the first part, I needed to implement the `__getitem__()` function in the CrackerBox dataset class. This function loads an image and its bounding box annotation, then processes them according to the YOLO format.

Here's my implementation:

def __getitem__(self, idx):

```
# gt file
filename_gt = self.gt_paths[idx]

# Get image file path (replace the -box.txt with .jpg)
filename_image = filename_gt.replace('-box.txt', '.jpg')

# Load image using OpenCV
image = cv2.imread(filename_image)

# Resize image to YOLO size (448x448)
image = cv2.resize(image, (self.yolo_image_size, self.yolo_image_size))

# Normalize pixels: subtract mean and divide by 255
image = image.astype(np.float32) - self.pixel_mean
image = image / 255.0

# Convert from (H,W,C) to (C,H,W) format for PyTorch
image = image.transpose((2, 0, 1))
image_blob = torch.from_numpy(image)

# Initialize ground truth tensors
gt_box_blob = torch.zeros(5, self.yolo_grid_num, self.yolo_grid_num)
gt_mask_blob = torch.zeros(self.yolo_grid_num, self.yolo_grid_num)
```

```
# Load ground truth bounding box (x1, y1, x2, y2)
with open(filename_gt, 'r') as f:
    box = f.readline().strip().split()
    x1 = float(box[0]) * self.scale_width
    y1 = float(box[1]) * self.scale_height
    x2 = float(box[2]) * self.scale_width
    y2 = float(box[3]) * self.scale_height

# Calculate center coordinates and width/height
cx = (x1 + x2) / 2
cy = (y1 + y2) / 2
w = x2 - x1
h = y2 - y1

# Determine which grid cell the center falls into
grid_x = int(cx / self.yolo_grid_size)
grid_y = int(cy / self.yolo_grid_size)

# Set the mask for this grid cell
gt_mask_blob[grid_y, grid_x] = 1

# Normalize box coordinates for the grid cell
# cx, cy: offset from the top-left corner of the grid cell, normalized to [0,1]
```

```
    cx_norm = (cx - grid_x * self.yolo_grid_size) / self.yolo_grid_size
    cy_norm = (cy - grid_y * self.yolo_grid_size) / self.yolo_grid_size

    # w, h: normalized by the image size to [0,1]
    w_norm = w / self.yolo_image_size
    h_norm = h / self.yolo_image_size

    # Store normalized values in the grid cell
    gt_box_blob[0, grid_y, grid_x] = cx_norm
    gt_box_blob[1, grid_y, grid_x] = cy_norm
    gt_box_blob[2, grid_y, grid_x] = w_norm
    gt_box_blob[3, grid_y, grid_x] = h_norm
    gt_box_blob[4, grid_y, grid_x] = 1.0  # confidence is 1 for ground truth

    # Return the sample dictionary
    sample = {'image': image_blob,
            'gt_box': gt_box_blob,
            'gt_mask': gt_mask_blob}

    return sample
```

I got the function working properly after a few tries. It basically:

1. Loads the image and resizes it to 448x448
2. Normalizes the pixel values
3. Converts the format for PyTorch (channels first)
4. Extracts the bounding box from the annotation file

5. Finds which grid cell contains the box center
6. Normalizes the box coordinates
7. Creates the output tensors

When I ran the data.py script, I could see the visualization of the original image, the image with grid cells and bounding box, and the mask showing which grid cell is responsible for detection.

Question 2: YOLO Network Implementation

Next, I implemented the YOLO network architecture based on the table in the assignment. Here's my implementation of the `create_modules()` function:

```
def create_modules(self):

    modules = nn.Sequential()


    Layer 1

    modules.add_module('Conv1', nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1))

    modules.add_module('ReLU1', nn.ReLU(inplace=True))

    modules.add_module('MaxPool1', nn.MaxPool2d(kernel_size=2, stride=2))


    Layer 2

    modules.add_module('Conv2', nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1))

    modules.add_module('ReLU2', nn.ReLU(inplace=True))

    modules.add_module('MaxPool2', nn.MaxPool2d(kernel_size=2, stride=2))


    Layer 3

    modules.add_module('Conv3', nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1))

    modules.add_module('ReLU3', nn.ReLU(inplace=True))

    modules.add_module('MaxPool3', nn.MaxPool2d(kernel_size=2, stride=2))
```

Layer 4

modules.add_module('Conv4', nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1))

modules.add_module('ReLU4', nn.ReLU(inplace=True))

modules.add_module('MaxPool4', nn.MaxPool2d(kernel_size=2, stride=2))

Layer 5

modules.add_module('Conv5', nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1))

modules.add_module('ReLU5', nn.ReLU(inplace=True))

modules.add_module('MaxPool5', nn.MaxPool2d(kernel_size=2, stride=2))

Layer 6

modules.add_module('Conv6', nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1))

modules.add_module('ReLU6', nn.ReLU(inplace=True))

modules.add_module('MaxPool6', nn.MaxPool2d(kernel_size=2, stride=2))

Layer 7

modules.add_module('Conv7', nn.Conv2d(512, 1024, kernel_size=3, stride=1, padding=1))

modules.add_module('ReLU7', nn.ReLU(inplace=True))

Layer 8

modules.add_module('Conv8', nn.Conv2d(1024, 1024, kernel_size=3, stride=1, padding=1))

modules.add_module('ReLU8', nn.ReLU(inplace=True))

Layer 9

```
    modules.add_module('Conv9', nn.Conv2d(1024, 1024, kernel_size=3, stride=1, padding=1))

    modules.add_module('ReLU9', nn.ReLU(inplace=True))


    # Flatten layer

    modules.add_module('Flatten', nn.Flatten())


    # Fully connected layers

    modules.add_module('FC1', nn.Linear(50176, 256))

    modules.add_module('FC2', nn.Linear(256, 256))


    # Output layer: 7×7×(5B+C) where B=2 boxes, C=1 class

    modules.add_module('FC_Output', nn.Linear(256, 7 * 7 * (5 * self.num_boxes +
self.num_classes)))


    # Sigmoid activation for the output

    modules.add_module('Sigmoid', nn.Sigmoid())


    return modules
```

The network structure follows the YOLO architecture from the paper, with:

- Convolutional layers with 3x3 kernels
- ReLU activations
- Max pooling layers to reduce spatial dimensions
- Three fully connected layers at the end
- A final sigmoid activation to get values between 0 and 1

When I ran model.py, it printed out the entire network architecture with all the layers and shapes, showing that my implementation was correct.

Question 3: Loss Function and Training

For the final part, I implemented the YOLO loss function:

```
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]

    # compute mask with shape (batch_size, num_boxes, 7, 7) for box assignment

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

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


    # compute assignment of predicted bounding boxes for ground truth bounding boxes

    for i in range(batch_size):

        for j in range(num_grids):

            for k in range(num_grids):


                # if the gt mask is 1

                if gt_mask[i, j, k] > 0:

                    # transform gt box

                    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
                # select the one with maximum IoU
                for b in range(num_boxes):
                    # center x, y and width, height
                    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))

    # compute yolo loss
    weight_coord = 5.0
    weight_noobj = 0.5

    # Loss on x coordinate (cx)
    loss_x = weight_coord * torch.sum(box_mask * torch.pow(gt_box[:, 0].unsqueeze(1) -
output[:, 0:5*num_boxes:5], 2.0))

    # Loss on y coordinate (cy)
    loss_y = weight_coord * torch.sum(box_mask * torch.pow(gt_box[:, 1].unsqueeze(1) -
output[:, 1:5*num_boxes:5], 2.0))
```

```
    # Loss on width (w)

    loss_w = weight_coord * torch.sum(box_mask * torch.pow(torch.sqrt(gt_box[:,
2].unsqueeze(1)) - torch.sqrt(output[:, 2:5*num_boxes:5]), 2.0))



    # Loss on height (h)

    loss_h = weight_coord * torch.sum(box_mask * torch.pow(torch.sqrt(gt_box[:,
3].unsqueeze(1)) - torch.sqrt(output[:, 3:5*num_boxes:5]), 2.0))



    # Loss on object confidence

    loss_obj = torch.sum(box_mask * torch.pow(box_confidence - output[:, 4:5*num_boxes:5],
2.0))



    # Loss on non-object confidence

    loss_noobj = weight_noobj * torch.sum((1 - box_mask) * torch.pow(0 - output[:,
4:5*num_boxes:5], 2.0))



    # Loss on class prediction (for multi-class detection, here we have just one class)

    box_cls_mask = torch.sum(box_mask, dim=1)

    box_cls_mask = box_cls_mask > 0

    loss_cls = torch.sum(box_cls_mask * torch.pow(1 - output[:, 5*num_boxes:], 2.0))



    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))



    # the total loss

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

```
    return loss
```

This loss function follows the one from the YOLO paper, which has several parts:

1. Coordinate loss (x, y, w, h) with higher weight (5.0)
2. Object confidence loss
3. Non-object confidence loss with lower weight (0.5)
4. Class prediction loss

For training, I ran the training script with these hyperparameters:

- Batch size: 2
- Learning rate: 1e-4
- Number of epochs: 100

The training went smoothly, with the loss dropping pretty quickly at first and then more gradually. After training, I ran the test script, which computes precision, recall, and average precision (AP).

Observations and Results

I got really good results! My final model achieved an AP of 0.61 on the validation set, which is double the required 30%. The precision-recall curve looks pretty good too, showing that the model has a good balance between finding real objects and avoiding false positives.

The training loss curve shows a nice convergence pattern. It starts around 10 and drops quickly in the first few epochs, then continues to decrease more gradually, reaching about 1 by epoch 30. This shows that the model was learning effectively throughout training.

![Loss curve showing convergence over epochs]

When I visualized some detections, the model correctly identified the cracker box in different positions and lighting conditions, which shows the effectiveness of the YOLO approach.

Conclusion

This YOLO implementation worked really well for the cracker box detection task. The AP of 0.61 far exceeds the required 30%, demonstrating the effectiveness of the single-shot detection approach.

I found several things challenging while working on this assignment:

1. Getting the grid cell assignments right for the ground truth
2. Making sure all the tensor shapes matched up correctly

3. Implementing the complex loss function with all its components

But overall, this was a great hands-on experience with object detection. I learned a lot about how YOLO works and how to implement neural networks in PyTorch. The assignment showed me the power of single-shot detection approaches and gave me practical experience with the full pipeline from data loading to evaluation.

If I had more time, I would experiment with data augmentation or transfer learning to see if I could push the performance even higher, but the current results are already pretty impressive.