CS 6384 Computer Vision Homework 5

Professor Yu Xiang

April 12, 2023

Download the homework5_programming.zip file from eLearning, Assignments, Homework 5. Finish the following programming problems and submit your scripts to eLearning. You can zip all the data and files for submission. Our TA will run your scripts to verify them.

Install the Python packages needed by

- pip install -r requirement.txt

Here are some useful resources:

- Python basics `https://pythonbasics.org/`
- Numpy `https://numpy.org/doc/stable/user/basics.html`
- OpenCV `https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html`
- PyTorch `https://pytorch.org/tutorials/`

In this homework, you will implement the YOLO object detector using PyTorch. The code can run on a CPU machine, no GPU is needed.

The implementation is based on the following paper:

- You Only Look Once: Unified, Real-Time Object Detection. Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi. In CVPR, 2016. `https://arxiv.org/abs/1506.02640`.

Problem 1

(3 points) DataLoader in PyTorch for YOLO.

Dataset. We provide a dataset of a cracker box to train the YOLO detector. The images and annotations are stored in yolo/data/. The input images are named as %06d.jpg. The ground truth bounding box locations are stored in %06d-box.txt. The four numbers in a txt file indicate the location of the cracker box in the corresponding image with the format (x1, y1, x2, y2), i.e., top-left and bottom-right corners of the bounding box. The images with names %06d-gt.jpg are only for visualization. We won't use these images in training and testing. We split the images in this dataset into 100 training images and 100 validation images.

Implement the __getitem__() function in yolo/data.py for the CrackerBox dataset class in PyTorch. The __getitem__ function loads and returns a sample from the dataset at the given index idx. Details about the dataset class in PyTorch can be found here: `https://pytorch.org/tutorials/beginner/basics/data_tutorial.html`.

In our case, the __getitem__ function should return a Python dictionary with three items.

- **sample['image']**: this should be a tensor in PyTorch with shape (3, 448, 448) to represent the input image. The original image size in the dataset is 480x640. So you need to resize it to 448x448. After resizing, we need to normalize the pixels by subtracting the pixel_mean (defined in the CrackerBox class) and then dividing by 255. Finally, tenors in PyTorch are stored with shape (channel, height, width). You need to swap the dimensions of the input image.

- **sample['gt_box']**: this should be a tensor in PyTorch with shape (5, 7, 7) to represent the ground truth bounding box. In this YOLO implementation, an input with size 448x448 are divided into 7x7 grids. If the center of the ground truth bounding box falls into a grid cell, that grid cell is responsible for detecting the object.

  In sample['gt_box'], each cell stores 5 values. They are $(cx, cy, w, h, confidence)$, where (cx, cy) are the center of the ground truth bounding box and (w, h) are the width and height of the ground truth bounding. For ground truth bounding boxes, the confidence is defined to be 1, i.e., $confidence = 1$.

  First of all, you can load the ground truth bounding boxes from the txt annotation files. Note that they are in (x1, y1, x2, y2) format. Since we scaled the input image to 448x448, you need to scale the bounding box as well. After that, we need to normalize (cx, cy, w, h) to $[0, 1]$. To do so, for (cx, cy), we compute its offset with respect to the top-left corner of the grid cell, and then divide the offset by the size of the cell (64 pixels in our case). In this way, the bounding box center can be normalized to $[0, 1]$. For width and height (w, h), we divide them by the size of the input image (448 pixels). For cells without objects, we store 0s in sample['gt_box'].

- **sample['gt_mask']**: this should be a tensor in PyTorch with shape (7, 7) to represent the location of the ground truth bounding in the grid. This tensor stores 0 and 1, where 1 indicates that the center of the ground truth bounding box falls into the corresponding cell.

After your implementation, run the yolo/data.py script in Python to verify it. The main function in this script creates a dataloader of the cracker box dataset and then visualizes the samples generated from the __getitem__ function. Figure 1 shows an example of running the script. Note that, each call of the __getitem__ function returns one sample. But PyTorch dataloader automatically groups multiple samples into a mini-batch by adding one more dimension to the tensors in the samples. For example, the image tensor will have size (batch_size, 3, height, width) from the dataloader.

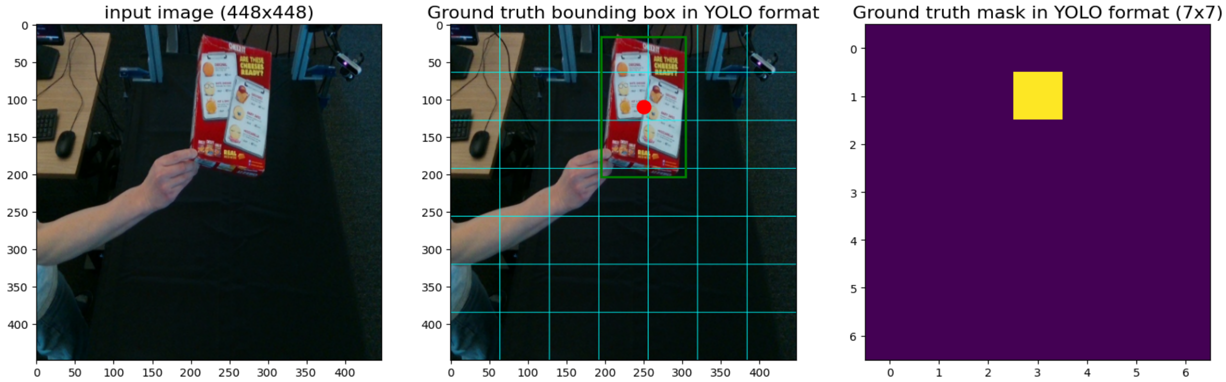


Figure 1: Visualization of data from the CrackerBox dataset class.

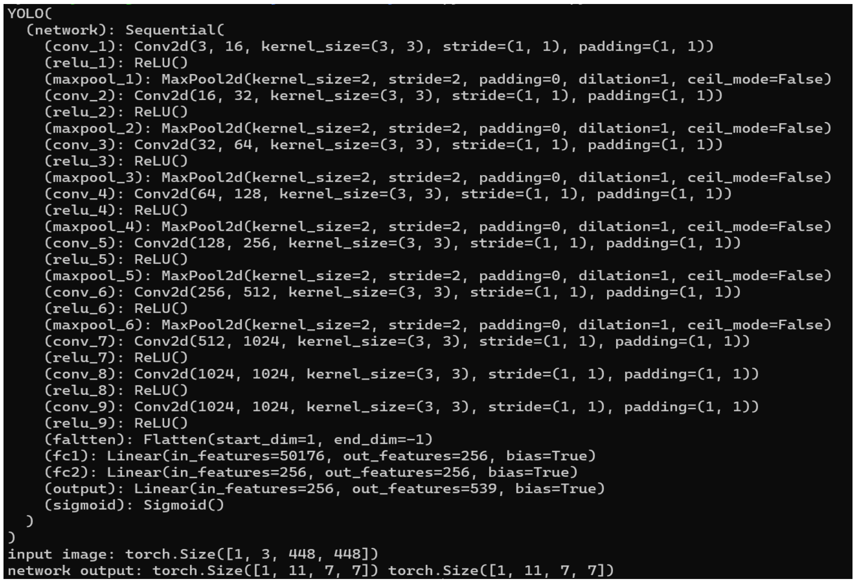
Problem 2

(3 points) Implement the YOLO network.

In this problem, you need to implement the YOLO network in yolo/model.py using the provided layer functions from PyTorch. Table 1 describes the detailed network architecture of the YOLO network. Follow this architecture and then implement the create_modules() function.

In this function, we first define the modules by **modules = nn.Sequential()**. Then you can use **modules.add_module()** to add layers to the modules. Refer to the PyTorch tutorial on building networks: `https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html`

After your implementation, run the yolo/model.py in Python to verify your network. Figure 2 shows a screenshot of running the script.



```
YOLO(
  (network): Sequential(
    (conv_1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu_1): ReLU()
    (maxpool_1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (conv_2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu_2): ReLU()
    (maxpool_2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (conv_3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu_3): ReLU()
    (maxpool_3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (conv_4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu_4): ReLU()
    (maxpool_4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (conv_5): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu_5): ReLU()
    (maxpool_5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (conv_6): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu_6): ReLU()
    (maxpool_6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (conv_7): Conv2d(512, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu_7): ReLU()
    (conv_8): Conv2d(1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu_8): ReLU()
    (conv_9): Conv2d(1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu_9): ReLU()
    (faltten): Flatten(start_dim=1, end_dim=-1)
    (fc1): Linear(in_features=50176, out_features=256, bias=True)
    (fc2): Linear(in_features=256, out_features=256, bias=True)
    (output): Linear(in_features=256, out_features=539, bias=True)
    (sigmoid): Sigmoid()
  )
)
input image: torch.Size([1, 3, 448, 448])
network output: torch.Size([1, 11, 7, 7]) torch.Size([1, 11, 7, 7])
```

Figure 2: Screenshot of running of the yolo/model.py script.

Layers	Dimensions (c, h, w)	Parameters
Input	3 × 448 × 448	
Conv1	16 × 448 × 448	kernel 3, stride 1, padding 1
ReLU1	16 × 448 × 448	
MaxPool1	16 × 224 × 224	kernel 2, stride 2
Conv2	32 × 224 × 224	kernel 3, stride 1, padding 1
ReLU2	32 × 224 × 224	
MaxPool2	32 × 112 × 112	kernel 2, stride 2
Conv3	64 × 112 × 112	kernel 3, stride 1, padding 1
ReLU3	64 × 112 × 112	
MaxPool3	64 × 56 × 56	kernel 2, stride 2
Conv4	128 × 56 × 56	kernel 3, stride 1, padding 1
ReLU4	128 × 56 × 56	
MaxPool4	128 × 28 × 28	kernel 2, stride 2
Conv5	256 × 28 × 28	kernel 3, stride 1, padding 1
ReLU5	256 × 28 × 28	
MaxPool5	256 × 14 × 14	kernel 2, stride 2
Conv6	512 × 14 × 14	kernel 3, stride 1, padding 1
ReLU6	512 × 14 × 14	
MaxPool6	512 × 7 × 7	kernel 2, stride 2
Conv7	1024 × 7 × 7	kernel 3, stride 1, padding 1
ReLU7	1024 × 7 × 7	
Conv8	1024 × 7 × 7	kernel 3, stride 1, padding 1
ReLU8	1024 × 7 × 7	
Conv9	1024 × 7 × 7	kernel 3, stride 1, padding 1
ReLU9	1024 × 7 × 7	
Flatten	50176	
FC1	256	
FC2	256	
FC Output	$7 \times 7 \times (5B + C)$	B: #boxes per cell, C: #classes
Sigmoid	$7 \times 7 \times (5B + C)$	

Table 1: YOLO network architecture.

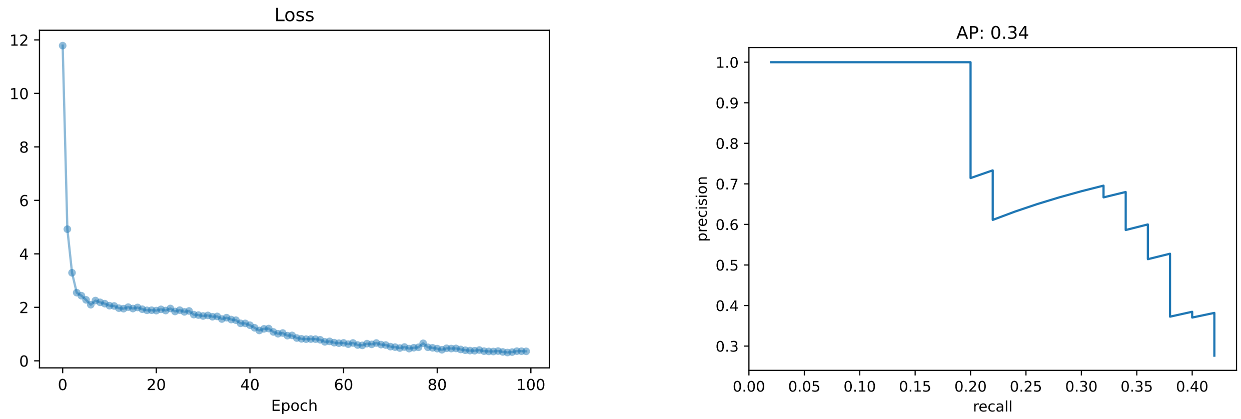
Problem 3

(4 points) Training and testing the YOLO network.

(3.1) Loss function. In order to train the YOLO network, we need to implement the loss function first. Figure 3 shows the loss function defined in the YOLO paper. Finish the compute_loss() function in yolo/loss.py to implement this loss function.

$$\begin{aligned}
&\lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
&+ \lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
&+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2
\end{aligned}$$

Figure 3: YOLO loss function.



(a) Training losses for 100 epochs.

(b) Precision-recall curve and Average Precision on the validation set.

Figure 4: Training and testing of the YOLO network for object detection.

In YOLO, each grid cell predicts multiple bounding boxes. In our case, each cell predicts 2 bounding boxes. In this compute_loss() function, we first find out which predicted bounding box is responsible for the ground truth bounding box. This is done by computing the Intersection over

Union (IoU) between the prediction and the ground truth. This assignment is stored in the box_mask tensor for future use, which corresponds to $\mathbb{1}_{ij}^{\textbf{obj}}$ in the loss function in Figure 3. Meanwhile, the IoUs are stored in the box_confidence tensor, which is treated as the target confidence scores the YOLO network should predict, i.e., C_i in the loss function.

(3.2) Training. After implementing the loss function, run the yolo/train.py script to train the network. You can tune the hyper-parameters defined in the beginning of the main function in this training script. Figure 4a shows the training loss over 100 epochs in our run. After training, the weights of the network will be saved to yolo/checkpoints/yolo_final.checkpoint.pth. The plot of the training losses will be saved to train_loss.pdf. You need to include this training loss plot in your homework submission.

(3.2) Testing. Test your trained model by running the yolo/test.py script. This script will create a dataloader for the validation set and run detection over all the validation images. In the end, it will compute recall, precision and average precision (AP), and plot the PR-curve as shown in Figure 4b. Include this plot in your submssion. **You need to achieve at least 30% AP for your trained model on the validation set, and the higher AP the better. You can tune these hyper-parameters for training if needed.** The test script also has a visualize() function to show the detection on the validation images. You can use it to visualize the detection results.