BME646 and ECE60146: Homework 6

Spring 2024
Due Date: 11:59pm, March 1, 2024
Checkpoint Due Date: 11:59pm, Feb 21, 2024
TA: Akshita Kamsali (akamsali@purdue.edu)

Turn in typed solutions via BrightSpace. Additional instructions can be found at BrightSpace. Late submissions will be accepted with penalty: **-10 points per-late-day, up to 5 days**. **This is a VERY challenging homework. Start early!**

1 Introduction

In HW5, you worked with skip connections to improve the performance of the Deep CNN classifier. Now we use these skip connections to construct an object detector network.

Your task in HW6 is to build your own object detector for *cakes* and other objects.

Because it is a more complex homework compared to what you have worked on so far, you are going to need more time for this. So we are going to help you pace your work by having you first submit a "Checkpoint" by Feb 21. Subsequently, the deadline for the homework will be March 1. You will only be allowed to upload your final submission if you submitted the checkpoint. The checkpoint will count for 10% of the overall grade for this homework.

1. For the Checkpoint due Feb 21, 2024, you will work on the problem of single instance object detection. **This will only require running a script from the Examples directory of DLStudio**. That is, you will NOT be writing any new code for the Checkpoint submission.

2. For the final homework solution that is due March 1, 2024, you will implement multi-instance object detection using YOLO.

Single-instance object detection is based on the assumption that each image has only one object of interest (even when the image contains multiple objects). The goal in single-instance object is to detect and localize that object. Localization is carried out by predicting the coordinates of the bounding box for the detected object.

Multi-instance object detection, on the other hand, is based on the assumption an image may contain multiple objects of interest and you want to predict their labels and the coordinates of their bounding boxes.

Before launching into this homework, make sure that you have downloaded the newly released Version 2.3.4 of DLStudio from

https://engineering.purdue.edu/kak/distDLS/DLStudio-2.3.4.html

It constains the updated versions of `SkipBlock` and the networks that use `SkipBlock` as a building block. You will be using one or more of those networks for the current homework.

2 Getting Ready for This Homework

2.1 For Submitting the Checkpoint

The main thing you have to do for submitting the checkpoint is to run the following script from the main Examples directory of DLStudio:

```
object_detection_and_localization.py
```

on the following training and testing datasets:

```
PurdueShapes5-10000-train.gz
PurdueShapes5-1000-test.gz
```

The integer value you see in the names of the datasets is the number of images in each. You can download these datasets by clicking on the link "*Download the image datasets for the main DLStudio module*" at the main webpage for DLStudio. This link will give you not only the above two datasets, but also the datasets you are going to need for some of the future homework assignments in our class.

After you have downloaded the data archive into the `Examples` directory in your installation of DLStudio, you would need to execute the following (Linux) commands

```
tar xvf datasets_for_DLStudio.tar.gz
```

This will create a subdirectory `data` in the `Examples` directory and deposit all the datasets in it.

That will set you up to execute the previously mentioned script `object_detection_and_localizati` `.py` in the Examples directory. This script uses the network class `LOADnet2`

from DLStudio. The acronym "LOAD" in "LOADnet2" stands for "LOcalization And Detection".

To understand the connection between the above mentioned script and the network class `LOADnet2` network class, search for the following string in the DLStudio.py file:

```
class DetectAndLocalize
```

As you will see, this class contains ALL of the DLStudio code dealing with single-instance object detection and localization. It defines multiple networks and difference loss functions for the job.

As you will see, the `LOADnet2` class is very much like the `BMEnet` class you saw in your previous homework. It uses the same `SkipBlock` that you are already familiar with. The main difference between `BMEnet` and `LOADnet2` is that the latter predicts both the class label for the detected object and estimates the coordinates of its bounding box. Predicting the coordinates is referred to as regression in DL.

For the Checkpoint submission, show the results you get with the DLStudio datasets mentioned above. The work you submit must also include a brief write-up on your understanding of the architecture of LOADnet network.

2.2 For the Final Submission of HW6

NOTE: It would be best if you read the material in this section after the class on Tuesday, Feb 20.

The very first thing you would need to do would be to beef up `SkipBlock` you used for the Checkpoint submission. In order to do that, it would be best if you first become familiar the logic of the ResNet as described in the paper:

https://arxiv.org/abs/1512.03385

and with the GitHub code for ResNet:

https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py

As you will see, ResNet has two different kinds of skip blocks, named `BasicBlock` and `BottleNeck`. `BasicBlock` is used as a building-block in ResNet-18 and ResNet-34. The numbers 18 and 34 refer to the number of layers in these two networks. For deeper networks, ResNet uses the `BottleNeck` class.

For the final submission, you will also be comparing two different loss functions for the regression loss: The L_2-norm based loss as provided by `torcn.nn.MSELoss` and the CIoU Loss as provided by PyTorch's `complete_box_iou_loss` that is available at the link supplied in the Intro section. To prepare for this comparison, review the material on Slides 38 through 48 of the Week 7 slides on Object Detection.

Your main goal for the final submission is to implement a multi-instance object detection framework with a YOLO network. The rest of this section details the steps you would need to go through for that.

1. Your first step would be to come to terms with the basic concepts of YOLO. As will be explained in the class next Tuesday, the YOLO logic is based on the notion of of Anchor Boxes. You divide an image into a grid of cells and you associate N anchor boxes with each cell in the grid. Each anchor box represents a bounding box with a different aspect ratio.

   Your first question is likely to be: **Why divide the image into a grid of cells?** To respond, the job of estimating the exact location of an object is assigned to that cell in the grid whose center is closest to the center of the object itself. Therefore, in order to localize the object, all that needs to be done is to estimate the offset between the center of the cell and the center of true bounding box for the object.

   But why have multiple anchor boxes at each cell of the grid? As previously mentioned, anchor boxes are characterized by different aspect ratios. That is, they are candidate bounding boxes with different height-to-width ratios. In Prof. Kak's implementation in the `RegionProposalGenerator` module, he creates five different anchor boxes for each cell in the grid, these being for the aspect ratios: `[1/5, 1/3, 1/1, 3/1, 5/1]`. The idea here is that the anchor box whose aspect ratio is closest to that of the true bounding box for the object will speak with the greatest confidence for that object.

2. You can deepen your understanding of the YOLO logic by looking at the implementation of image gridding and anchor boxes in Version 2.1.1 of Prof. Kak's `RegionProposalGenerator` module:

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

   Go to the Example directory and execute the script:

```
multi_instance_object_detection.py
```

and work your way backwards into the module code to see how it works. In particular, you should pay attention to how the notion of anchor boxes is implemented in the function:

```
run_code_for_training_multi_instance_detection()
```

To execute the script `multi_instance_object_detection.py`, you will need to download and install the following datasets:

```
Purdue_Dr_Eval_Multi_Dataset-clutter-10-noise-20-size-10000-train.gz
Purdue_Dr_Eval_Multi_Dataset-clutter-10-noise-20-size-1000-test.gz
```

Links for downloading the datasets can be found on the module's webpage. In the dataset names, a string like `size-10000` indicates the number of images in the dataset, the string `noise-20` means 20% added random noise, and the string `clutter-10` means a maximum of 10 background clutter objects in each image.

Follow the instructions on the main webpage for `RegionProposalGenerator` on how to unpack the image data archive that comes with the module and where to place it in your directory structure. These instructions will ask you to download the main dataset archive and store it in the `Examples` directory of the distribution.

3 Programming Tasks

3.1 Checkpoint Submission

Checkpoint submission should NOT require any programming by you. All you have to do is to run the DLStudio script as described in Section 2.1 and submit your results. The document you submit should include a brief writup on your understanding of the `LOADnet` network.

3.2 Final Submission of HW6

This section contains guidelines on how to extract images with one or more than one instance of the object from the COCO dataset. Finally, implement the YOLO logic to perform multi-instance detection.

3.2.1 How to Use the COCO Annotations

For this homework, you will need bounding boxes in addition to the labels from the COCO dataset. In this section, we go over how to access these annotations as shown in Fig. 1. The code below is sufficient to introduce you how to prepare your own dataset and write your dataloader for this homework.

Before we jump into the code, it is important to understand structures of the COCO annotations. The COCO annotations are stored in the list of dictionaries and what follows is an example of such a dictionary:

```
{
    "id": 1409619,                  # annotation ID
    "category_id": 1,               # COCO category ID
    "iscrowd": 0,                   # specifies whether the
                                    segmentation is for a single
                                    object or for a group/cluster
                                    of objects
    "segmentation": [
        [86.0, 238.8, ..., 382.74, 241.17]
    ],                              # a list of polygon vertices
                                    around the object (x, y pixel
                                    positions)
    "image_id": 245915,             # integer ID for COCO image
    "area": 3556.2197000000015,     # Area measured in pixels
    "bbox": [86, 65, 220, 334]      # bounding box [top left x
                                    position, top left y position,
                                    width, height]
}
```

The following code (refer to inline code comments for details) shows how to access the required COCO annotation entries and display a randomly chosen image with desired annotations for visual verification. After importing the required python modules (*e.g.* `cv2`, `skimage`, `pycocotools`, etc.), you can run the given code and visually verify the output yourself.

```
# Input
input_json = 'instances_train2017.json'
class_list = ['cake', 'dog', 'motorcycle']

##########################
# Mapping from COCO label to Class indices
coco_labels_inverse = {}
coco = COCO(input_json)
catIds = coco.getCatIds(catNms=class_list)
categories = coco.loadCats(catIds)
categories.sort(key=lambda x: x['id'])
```

```
print(categories)


for idx, in_class in enumerate(class_list):
    for c in categories:
        if c['name'] == in_class:
            coco_labels_inverse[c['id']] = idx
print(coco_labels_inverse)

############################
# Retrieve Image list
imgIds = coco.getImgIds(catIds=catIds)

############################
# Display one random image with annotation
idx = np.random.randint(0, len(imgIds))
img = coco.loadImgs(imgIds[idx])[0]
I = io.imread(img['coco_url'])
# change from grayscale to color
if len(I.shape) == 2:
    I = skimage.color.gray2rgb(I)
# pay attention to the flag, iscrowd being set to False
annIds = coco.getAnnIds(imgIds=img['id'], catIds=catIds,
                                    iscrowd=False)
anns = coco.loadAnns(annIds)
fig, ax = plt.subplots(1, 1)
image = np.uint8(I)
for ann in anns:
    [x, y, w, h] = ann['bbox']
    label = coco_labels_inverse[ann['category_id']]
    image = cv2.rectangle(image, (int(x), int(y)), (int(x + w),
                                     int(y + h)), (36, 255, 12), 2)
    image = cv2.putText(image, class_list[label], (int(x), int(
                                    y - 10)), cv2.
                                    FONT_HERSHEY_SIMPLEX,
                           0.8, (36, 255, 12), 2)
ax.imshow(image)
ax.set_axis_off()
plt.axis('tight')
plt.show()
```

3.2.2 Creating Your Own Multi-Instance Object Localization Dataset

In this exercise, you will create your own dataset based on following steps:

1. You need to write a script similar to HW4 that filters through the images and annotations to generate your training and testing dataset such that any image in your dataset meets the following criteria:

- Contains at least one *foreground object*. A foreground object must be from one of the three categories: `['cake', 'dog', 'motorcycle']`.

  Additionally, the area of any foreground object must be larger than $64 \times 64 = 4096$ pixels[1]. Different from the HW4, there can be multiple foreground objects in an image since we are dealing with multi-instance object localization for this homework.

  If there is none, that image should be discarded.

- When saving your images to disk, resize them to 256×256. Note that you would also need to scale the bounding box parameters accordingly after resizing.
- Again, use images from **2017 Train images** for the training set and **2017 Val images** for the testing set.

Figure 1: Sample COCO images with bounding box and label annotations for multi-instances.

Again, you have total freedom on how you organize your dataset as long as it meets the above requirements. If done correctly, you will end up with approximately 8000 train images and 300 test images.

2. In your report, make a figure of a selection of images from your created dataset. You should plot at least 3 images from each of the three classes like what is shown in Fig. 1 and with the annotations of all the present foreground objects.

[1] Also, you can use the area entry in the annotation dictionary instead of calculating it yourself.

3.2.3 Building Your Deep Neural Network

1. Once you have prepared the dataset, you now need to implement your deep convolutional neural network (CNN) for multi-instance object classification and localization. You can directly base your CNN architecture on LOADnet2 adjusting for YOLO parameters. Again, you have total freedom on what specific architecture you choose. You will need to use a beefed up `SkipBlock` in 2.2.

2. The key design choice you'll need to make is on the organization of the predicted parameters by your network. As you have learned in Prof. Kak's tutorial on Multi-Instance Object Detection [1], for any input image, your CNN should output a `yolo_tensor`.

3. The exact shape of your predicted `yolo_tensor` is dependent on how you choose to implement image gridding and the anchor boxes. It is highly recommended that, before starting your own implementation, you should review the tutorial again and familiarize yourself with the notions of `yolo_vector`, which is predicted for each and every anchor box, and `yolo_tensor`, which stacks all `yolo_vectors`.

4. In your report, designate a code block for your network architecture.

5. Additionally, clearly state the shape of your output `yolo_tensor` and explain in detail how that shape is resulted from your design parameters, *e.g.* the total number of cells and the number of anchor boxes per cell, etc.

3.3 Training and Evaluating Your Network

Now that you have finished designing your deep CNN, it is finally time to put your glorious *multi-cake* detector in action. What is described in this section is probably the most challenging part of the homework. To train and evaluate your YOLO framework, you should follow the steps below:

1. Write your own dataloader. While everyone's implementation will differ, it is *highly* recommended that the following items should be returned by your `__getitem__` method for multi-instance object localization:

   (a) The image tensor;

   (b) For each foreground object present in the image:

i. Index of the assigned cell;
ii. Index of the assigned anchor box;
iii. Groundtruth `yolo_vector`.

The tricky part here is how to best assign a cell and an anchor box given a GT bounding box. For this part, you will have to implement your own logic. Typically, one would start with finding the best cell, and subsequently, choose the anchor box with the highest IoU with the GT bounding box. You would need to pass on the indices of the chosen cell and anchor box for the calculation of the losses explained later in this section.

It is also worthy to remind yourself that the part in a `yolo_vector` concerning the bounding box should contain four parameters: δ_x, δ_y, σ_w and σ_h. The first two, δ_x and δ_y, are simply the offsets between the GT box center and the anchor box center. While the last two, σ_w and σ_h, can be the "ratios" between the GT and anchor widths and heights:

$$w_{\text{GT}} = e^{\sigma_w} \cdot w_{\text{anchor}},$$

$$h_{\text{GT}} = e^{\sigma_h} \cdot h_{\text{anchor}}.$$

2. Create your own training code (or adjust existing code) for training your network. This time, you'll need three different types of losses: a binary cross-entropy loss for detecting objects, a cross-entropy loss for classifying objects, and another loss for refining bounding box positions.

3. Develop your own evaluation code (or make modifications to existing code). Evaluating single-instance detectors, assessing the performance of a multi-instance detector can be more complex and may be beyond the scope of this homework assignment, as discussed in Prof. Kak's tutorial [1]. Therefore, for this assignment, we only require you to present your multi-instance detection and localization results in a qualitative manner. This means that for each test image, you should display the predicted bounding boxes and their corresponding class labels alongside the ground truth annotations.

   Specifically, you'll need to create your own method for translating the predicted `yolo_tensor` into bounding box predictions and class label predictions that can be visually represented. You have the flexibility to implement this logic according to your own approach.

4. In your report, write several paragraphs summarizing on how you have implemented your dataloading, training and evaluation logic.

   Additionally, include a plot of all three losses over training iterations (you should train your network for at least 10-15 epochs).

   For presenting the outputs of your YOLO detector, display your multi-instance localization and detection results for at least 8 different images from the *test set*. Again, for a given test image, you should plot the predicted bounding boxes and class labels along with the GT annotations for all foreground objects. You should strive to present your best *multi-instance* results in at least 6 images while you can use the other 2 images to illustrate the current shortcomings of your multi-instance detector. Additionally, you should include a paragraph that discusses the performance of your YOLO detector.

4 Submission Instructions

Include a typed report explaining how did you solve the given programming tasks. For checkpoint I and the final submission for HW6, you need to submit the following:

1. Your pdf must include a description of
   - The figures and descriptions as mentioned in Sec. 3.
   - For final submission only, your source code. Make sure that your source code files are adequately commented and cleaned up.
2. Turn in a pdf file a typed self-contained pdf report with source code and results. Rename your .pdf file as hw6_<First Name><Last Name>.pdf
3. Turn in a zipped file, it should include all source code files (only .py files are accepted). Rename your .zip file as hw5_<First Name><Last Name>.zip .
4. There will be seperate submission links for Checkpoint I and HW6 on Brightspace
5. **Do NOT submit your network weights.**
6. **Do NOT submit your dataset.**

7. For all homeworks, you are encouraged to use `.ipynb` for development and the report. If you use `.ipynb`, please convert it to `.py` and submit that as source code.

8. You can resubmit a homework assignment as many times as you want up to the deadline. Each submission will overwrite any previous submission. **If you are submitting late, do it only once on BrightSpace.** Otherwise, we cannot guarantee that your latest submission will be pulled for grading and will not accept related regrade requests.

9. The sample solutions from previous years are for reference only. **Your code and final report must be your own work.**

10. To help better provide feedback to you, make sure to **number your figures and tables**.

References

[1] Multi-Instance Object Detection – Anchor Boxes and Region Proposals. URL https://engineering.purdue.edu/DeepLearn/pdf-kak/MultiDetection.pdf.