CIS680: Vision and Learning
Project 3: Instance Segmentation: FPN & SOLO
Due: 2020 at 11:59 pm

1 Updates after release

- add dataloader output dimension.
- Architecture add Sigmoid followed last CNN: previously without this activation.
- Architecture last layer setting for both category and mask branch, cate branch changed to 3x3x(C-1), mask branch to$(num_grid)^2$. And some detail illustration: previously both are 3x3x3 which is wrong.
- the whole dataset is of 3265 images: previously is mistakely stated as training set.
- Architecture: mask branch's output layer CNN, there is no padding(pad=0): previously padding=1.
- delete img: (bz, 3, 300, 400) in session 4.3 mini-batch data structure. That should be img: (bz, 3, 800, 1088).
- session 6.4.1 forward, ins_pred_list: list, len(fpn_level), each (bz, S^2, Ori_H/4, Ori_W/4)
- Matrix NMS example is updated.

2 Instructions

- This is a group assignment of 2. Please start really early for this project.
- The whole project could be divided into 5 .py files:
  - *dataset.py*,
  - *backbone.py*,
  - *solo_head.py*,
  - *main_train.py* / *resume_train.py*,
  - *main_infer.py*.
- My advice would be finish every component in local machine, debug them. Then move them into colab session for training. You could upload all the .py files and use "! main_train.py" in colab for training, and save every epochs.
- the ideal training process will be around 6 hours.
- There is no single answer to most problems in deep learning, therefore the questions will often be underspecified. You need to fill in the blanks and submit a solution that solves the (practical) problem. Document the choices (hyperparameters, features, neural network architectures, etc.) you made in the write-up.
- All the code should be written in Python. You should use PyTorch only to complete this homework.

3 Overview

3.1 Instance segmentation

Instance segmentation task, different with the previous detection task, requires a combination of both object detection and semantic segmentation tasks as in Fig.1.

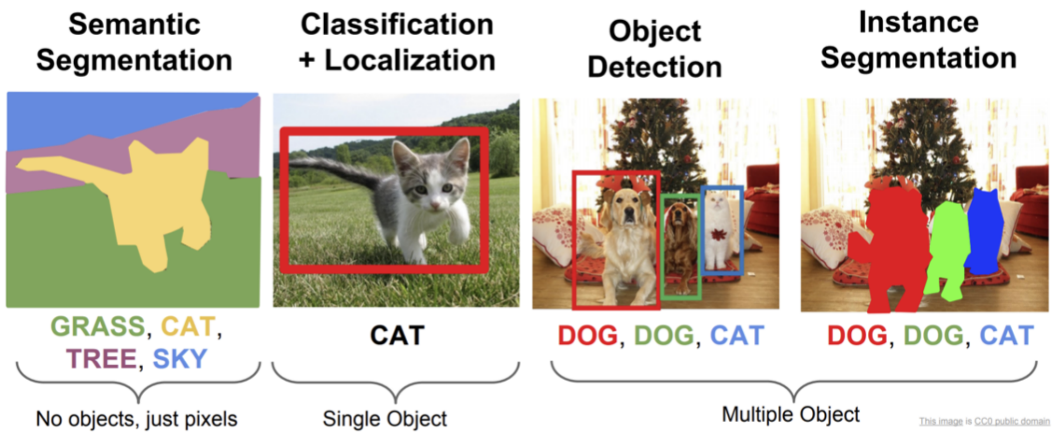


Figure 1: Instance Segmentation Task

3.2 SOLO intro

In this project we will implement a newly proposed instance segmentation framework, namely Segmenting Objects by Locations (SOLO). Like previous YOLO, SOLO is also a grid based framework but proposed to handle instance segmentation task. Different with traditional instance segmentation framework, SOLO does not propose the bounding box location, instead, it proposes instance category and instance segmentation mask directly.

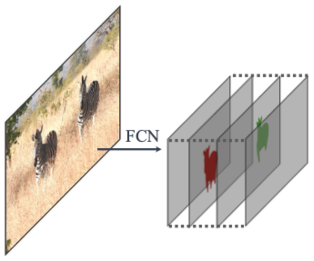


Figure 2: SOLO[?]

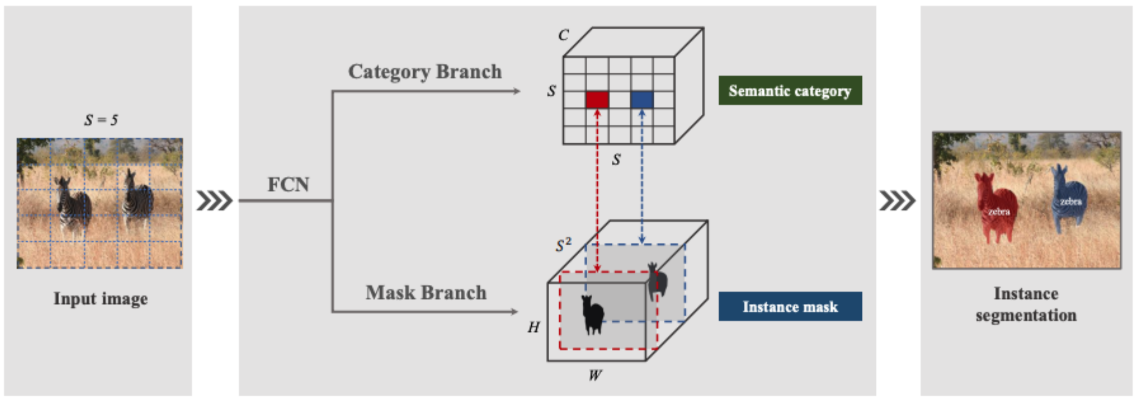


Figure 3: SOLO branches[?]

3.3 FPN intro

Feature pyramid Networks [?] are a commonly used component in recognition systems for detecting objects at different scales. Image forwards through backbone will go through several stages. At the last layer of each of stage, backbone will generate different size feature map. FPN use top-down architecture with lateral connections structure to build a series of high-level semantic feature maps at all scales.

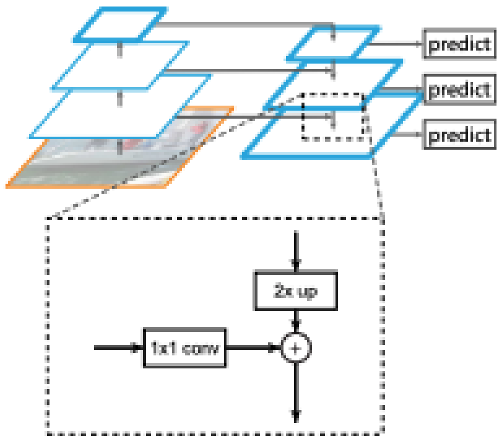


Figure 4: FPN[?]

As in this project, we will focus at how to integrate FPN into our network which is very important for boosting the performance of instance segmentation task.

4 Dataset Construction

4.1 Dataset description

Here is the dataset: instance segmentation dataset
In this assignment your task is to detect 3 types of objects: Vehicles, People and Animals.Your dataset consists of:
(1) a numpy array of all the RGB Images ($3 \times 300 \times 400$)
(2) a numpy array of all the masks (300×400)
(3) list of ground truth labels per image.
(4) list of ground truth bounding boxes per image. The four numbers are the upper left and lower right coordinates.
You should use the labels list to figure out which mask belongs to which image.

4.2 Part - A

(until solo_head.py Loss function)

4.3 dataset.py

In this .py file, you should implement dataset and dataloader class. Make 80% of whole data as the training data. There should be in total 3265 images in whole dataset. With its mask, category, boundingbox annotations.
class BuildDataset(torch.utils.data.Dataset)

This class should build dataset containing: images, labels, masks, bboxes for each of the image. First, take a look at the original numpy array to have a understanding of dataset. Masks are not saved w.r.t. each image, so you have to link masks to image. This grouping process could be done in the "__*init*__" function in class.

After grouping the masks into each image, we want to do a preprocess to the image:

- normalize image pixel value to [0,1].

- rescale the image to 800*1066,
  using torch.nn.functional.interpolate
- normalize each channel with mean:[0.485, 0.456, 0.406], std:[0.229, 0.224, 0.225],
  using torchvision.transforms.functional.normalize
- zero padding the image to 800*1088,
  using torch.nn.functional.pad

class BuildDataLoader(torch.utils.data.DataLoader)

in training, each time we have to make a batch of training data. Since different image contains different number of objects, we have to rewrite the default '*collect_fn*()' function in dataloader. Here is a sample code of '*collect_fn*()' function:

```
    def collect_fn(self, batch):
        transed_img_list = []
        label_list = []
        transed_mask_list = []
        transed_bbox_list = []

        for transed_img, label, transed_mask, transed_bbox in batch:
            transed_img_list.append(transed_img)
            label_list.append(label)
            transed_mask_list.append(transed_mask)
            transed_bbox_list.append(transed_bbox)

        return torch.stack(transed_img_list, dim=0),\
               label_list,\
               transed_mask_list,\
               transed_bbox_list
```

the mini-batch data structure will like following:

```
# output:
    # img: (bz, 3, 800, 1088)
    # label_list: list, len:bz, each (n_obj,)
    # transed_mask_list: list, len:bz, each (n_obj, 800,1088)
    # transed_bbox_list: list, len:bz, each (n_obj, 4)
```

Visual debugging for dataset.py
To make sure this part of the code is working, contain a *main* function which will visualize the image, instance bounding box, instance segmentation. E.g.

Remember to save 5 images as like above example dataset visualization.

5 backbone.py

In this .py file we use a Resnet50 directly from torchvision. Backbone code will directly give to you guys.

6 solo_head.py

solo_head.py will be divided into the following functions:

- '*__init__*', '*_init_layers*', '*_init_weights*'
- '*NewFPN*'
- '*MultiApply*'
- '*target*', '*target_single_img*'

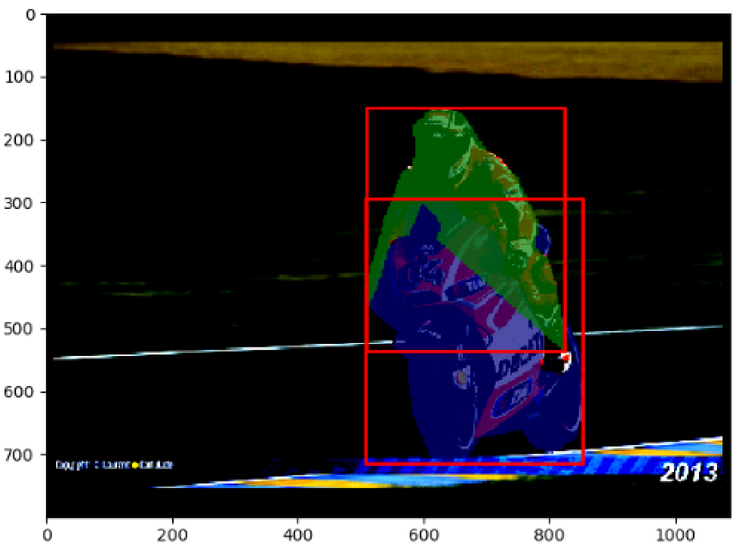


Figure 5: dataset visual 1

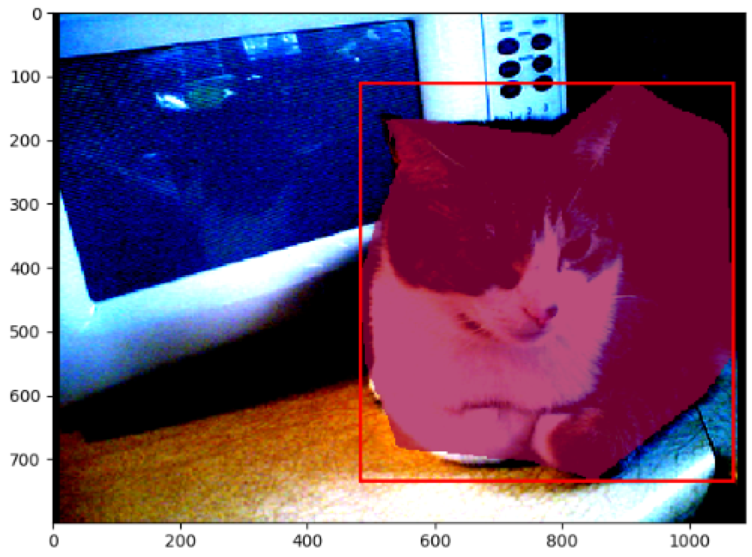


Figure 6: dataset visual 2

- ’*forward*’, ’*forward_single_level*’
- ’*PlotGT*’
- ’*loss*’, *DiceLoss*, *FocalLoss*
- ’*PostProcess*’, ’*PostProcessImg*’, ’*MartixNMS*’, ’*points_nms*’
- ’*PlotInfer*’

6.1 Model Initialization

6.1.1 __init__

All the default parameters are given as follows:

```
def __init__(self,
                 num_classes,
                 in_channels=256,
                 seg_feat_channels=256,
                 stacked_convs=7,
                 strides=[8, 8, 16, 32, 32],
                 scale_ranges=((1, 96), (48, 192), (96, 384), (192, 768), (384, 2048)),
                 epsilon=0.2,
                 num_grids=[40, 36, 24, 16, 12],
                 cate_down_pos=0,
                 with_deform=False,
                 mask_loss_cfg=dict(weight=3),
                 cate_loss_cfg=dict(gamma=2,
                                 alpha=0.25,
                                 weight=1),
                 postprocess_cfg=dict(cate_thresh=0.2,
                                       ins_thresh=0.5,
                                       pre_NMS_num=50,
                                       keep_instance=5,
                                       IoU_thresh=0.5))
```

In this function, you need to save all those parameters into self.variable. Then call function ’*_init_layers*’, ’*_init_weights*’ to build network and initialize its parameters.

6.1.2 _init_layers

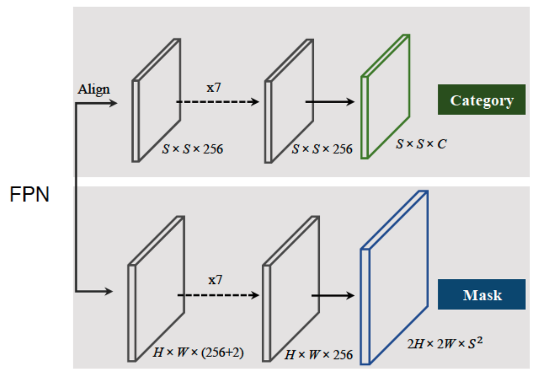


Figure 7: SOLO branches

Category branch

Layer	Hyperparameters
conv1	Kernel size = 3x3x256, stride = 1, pad = 1, bias = False. Followed by GroupNorm, num_groups=32 and ReLU
conv2	Kernel size = 3x3x256, stride = 1, pad = 1, bias = False. Followed by GroupNorm, num_groups=32 and ReLU
conv3	Kernel size = 3x3x256, stride = 1, pad = 1, bias = False. Followed by GroupNorm, num_groups=32 and ReLU
conv4	Kernel size = 3x3x256, stride = 1, pad = 1, bias = False. Followed by GroupNorm, num_groups=32 and ReLU
conv5	Kernel size = 3x3x256, stride = 1, pad = 1, bias = False. Followed by GroupNorm, num_groups=32 and ReLU
conv6	Kernel size = 3x3x256, stride = 1, pad = 1, bias = False. Followed by GroupNorm, num_groups=32 and ReLU
conv7	Kernel size = 3x3x256, stride = 1, pad = 1, bias = False. Followed by GroupNorm, num_groups=32 and ReLU
conv_out	Kernel size = 3x3x(C-1), pad=1, bias = True. Followed by Sigmoid layer. C here is 4 in our case, including background as a convention.

Mask branch

Layer	Hyperparameters
conv1	Kernel size = 3x3x256 (input channel: 256+2, coordinate convolution will be illustrate in forward) stride = 1, pad = 1, bias=False. Followed by GroupNorm, num_groups=32 and ReLU
conv2	Kernel size = 3x3x256, stride = 1, pad = 1, bias = False. Followed by GroupNorm, num_groups=32 and ReLU
conv3	Kernel size = 3x3x256, stride = 1, pad = 1, bias = False. Followed by GroupNorm, num_groups=32 and ReLU
conv4	Kernel size = 3x3x256, stride = 1, pad = 1, bias = False. Followed by GroupNorm, num_groups=32 and ReLU
conv5	Kernel size = 3x3x256, stride = 1, pad = 1, bias = False. Followed by GroupNorm, num_groups=32 and ReLU
conv6	Kernel size = 3x3x256, stride = 1, pad = 1, bias = False. Followed by GroupNorm, num_groups=32 and ReLU
conv7	Kernel size = 3x3x256, stride = 1, pad = 1, bias = False. Followed by GroupNorm, num_groups=32 and ReLU
conv_out	Kernel size = 1x1x$(num_grid)^2$, bias = True. Followed by Sigmoid layer notice last layer is a list of layers because *num_grid* is different for each FPN level.

One small tip: since we have repeat stack of layers, you could use nn.ModuleList() and for loop to include all the layer.

6.1.3 _init_weights

For all the convolution kernel weight we use *nn.init.xavier_uniform_* as the initialization. And for the bias term we use 0 as initialization.
refer to https://pytorch.org/docs/stable/nn.init.html for initialization.

6.2 NewFPN

```
# Input:
    # fpn_feat_list, list, len(FPN), stride[4,8,16,32,64]
# Output:
    # new_fpn_list, list, len(FPN), stride[8,8,16,32,32]
```

This function is to reshape feature pyramids. Original stride of ResNet50 is [4,8,16,32,64], in order to save space when running model, we resize the first level to 1/2. And following paper setting resize the last level to 2. We get stride [8,8,16,32,32]. Use *'torch.nn.functional.interpolate'* to resize feature map.

6.3 MultiApply

This function will be used in used in '*target*' and '*forward*'. It is a map function which utily the CPU parallel computing. In '*target*', it is used to build y_label simultaneously for every image in a mini-batch. In '*forward*', it is used to forward the feature simultaneously for every feature pyramid.

This function will be provided as follows:

```
def MultiApply(self, func, *args, **kwargs):
    pfunc = partial(func, **kwargs) if kwargs else func
    map_results = map(pfunc, *args)

    return tuple(map(list, zip(*map_results)))
```

6.4 Parallel Forward Process

6.4.1 forward

```
# Forward function should forward every levels in the FPN.
# this is done by map function or for loop
# Input:
    # fpn_feat_list: backout_list of resnet50-fpn
# Output:
    # if eval = False
        # cate_pred_list: list, len(fpn_level), each (bz,C-1,S,S)
        # ins_pred_list: list, len(fpn_level), each (bz, S^2, 2H_feat, 2W_feat)
    # if eval==True
        # cate_pred_list: list, len(fpn_level), each (bz,S,S,C-1)
        #/ after point_NMS
        # ins_pred_list: list, len(fpn_level), each (bz, S^2, Ori_H/4, Ori_W/4)
        #/ after upsampling
```

As a hint, in this function we use *MultiApply* to utilize CPU parallel computing:

```
cate_pred_list, ins_pred_list = \
    self.MultiApply(self.forward_single_level,
                    new_fpn_list,
                    list(range(len(new_fpn_list))),
                    eval=eval, upsample_shape=quart_shape)
```

6.4.2 forward_single_level

```
# This function forward a single level of fpn_featmap through the network
# Input:
    # fpn_feat: (bz, fpn_channels(256), H_feat, W_feat)
    # idx: indicate the fpn level idx, num_grids idx, the ins_out_layer idx
# Output:
    # if eval==False
        # cate_pred: (bz,C-1,S,S)
        # ins_pred: (bz, S^2, 2H_feat, 2W_feat)
    # if eval==True
        # cate_pred: (bz,S,S,C-1) / after point_NMS
        # ins_pred: (bz, S^2, Ori_H/4, Ori_W/4) / after upsampling
```

Two spetial settings for each branch:

for category branch, we resize feature map use torch.nn.functional.interpolate, make resolution to $S \times S$, where S is number of grid. Notice for different feature pyramid level, we have different number of grid, defined in '*__init__*' for each feature pyramid(from high resolution to low resolution)

for mask branch, we concatenate a coordinate convolution channel which is adding pixel normalized x,y coordinate information to the feature map we get from backbone. Then we will have 256+2 channels as input to mask branch.
Notice here, behavior of forward is slightly different when eval==True.

6.5 Build Target

6.5.1 target

```
# This function build the ground truth tensor for each batch in the training
# Input:
    # ins_pred_list: list, len(fpn_level), each (bz, S^2, 2H_feat, 2W_feat)
    # / ins_pred_list is only used to record feature map
    # bbox_list: list, len(batch_size), each (n_object, 4) (x1y1x2y2 system)
    # label_list: list, len(batch_size), each (n_object, )
    # mask_list: list, len(batch_size), each (n_object, 800, 1088)
# Output:
    # ins_gts_list: list, len(bz), list, len(fpn), (S^2, 2H_f, 2W_f)
    # ins_ind_gts_list: list, len(bz), list, len(fpn), (S^2,)
    # cate_gts_list: list, len(bz), list, len(fpn), (S, S), {1,2,3}
```

Here we build ground truth tensor, which we later used for learning. Notice here the resolution of ins_gt (mask branch ground truth) and ins_pred(mask branch prediction) is 2×[H_feat, W_feat] feature pyramid resolution of that level.

ins_ind_gts_list records the activated channel in the mask branch prediction which comes in handle in loss computation.

6.5.2 target_single_img

Notice here we follow the row-based indexing for channels, that is, if we flatten $S \times S$ grids to S^2 channels, then (i, j) grids corresponding to $i \times S + j$ channels.

```
# input:
    # gt_bboxes_raw: n_obj, 4 (x1y1x2y2 system)
    # gt_labels_raw: n_obj,
    # gt_masks_raw: n_obj, H_ori, W_ori
    # featmap_sizes: list of shapes of featmap
# output:
    # ins_label_list: list, len: len(FPN), (S^2, 2H_feat, 2W_feat)
    # cate_label_list: list, len: len(FPN), (S, S)
    # ins_ind_label_list: list, len: len(FPN), (S^2, )
```

1. $cate_{label} = [S, S]$, $ins_{label} = [S^2, 2H_{feat}, 2W_{feat}]$. for each FPN level.
2. each level in FPN corresponding to a different grid cut image into S by S conceptually.
3. instance scale computed as $\sqrt{wh}$ in FPN level assignment. In author's code w, h is computed w.r.t. bbox.
4. object center region: $(c_x, c_y, \varepsilon w, \varepsilon h)$ where c_x, c_y is the mass center, the ε is set as 0.2.
5. $grid(i, j)$ is positive if it falls into center region of an object. How this work in code is finding the top, bottom, left, right grid activate them with category label.
   $k = i \cdot S + j$ to assign the channel in the ins_{label} which is the mask ground truth labelling.
   one grid $\rightarrow$ one object, one object $\rightarrow$ multiple grids.

6.6 PlotGT

```
# this function visualize the ground truth tensor
# Input:
    # ins_gts_list: list, len(bz), list, len(fpn), (S^2, 2H_f, 2W_f)
    # ins_ind_gts_list: list, len(bz), list, len(fpn), (S^2,)
    # cate_gts_list: list, len(bz), list, len(fpn), (S, S), {1,2,3}
    # color_list: list, len(C-1)
    # img: (bz,3,Ori_H, Ori_W)
    ## self.strides: [8,8,16,32,32]
```

This is a visual debugging session before we move to Loss session, for each image, this function plot out 5 masked images to check if ground truth mask is assigned correctly in each of feature pyramid.
Following are examples Fig.[8][9]:

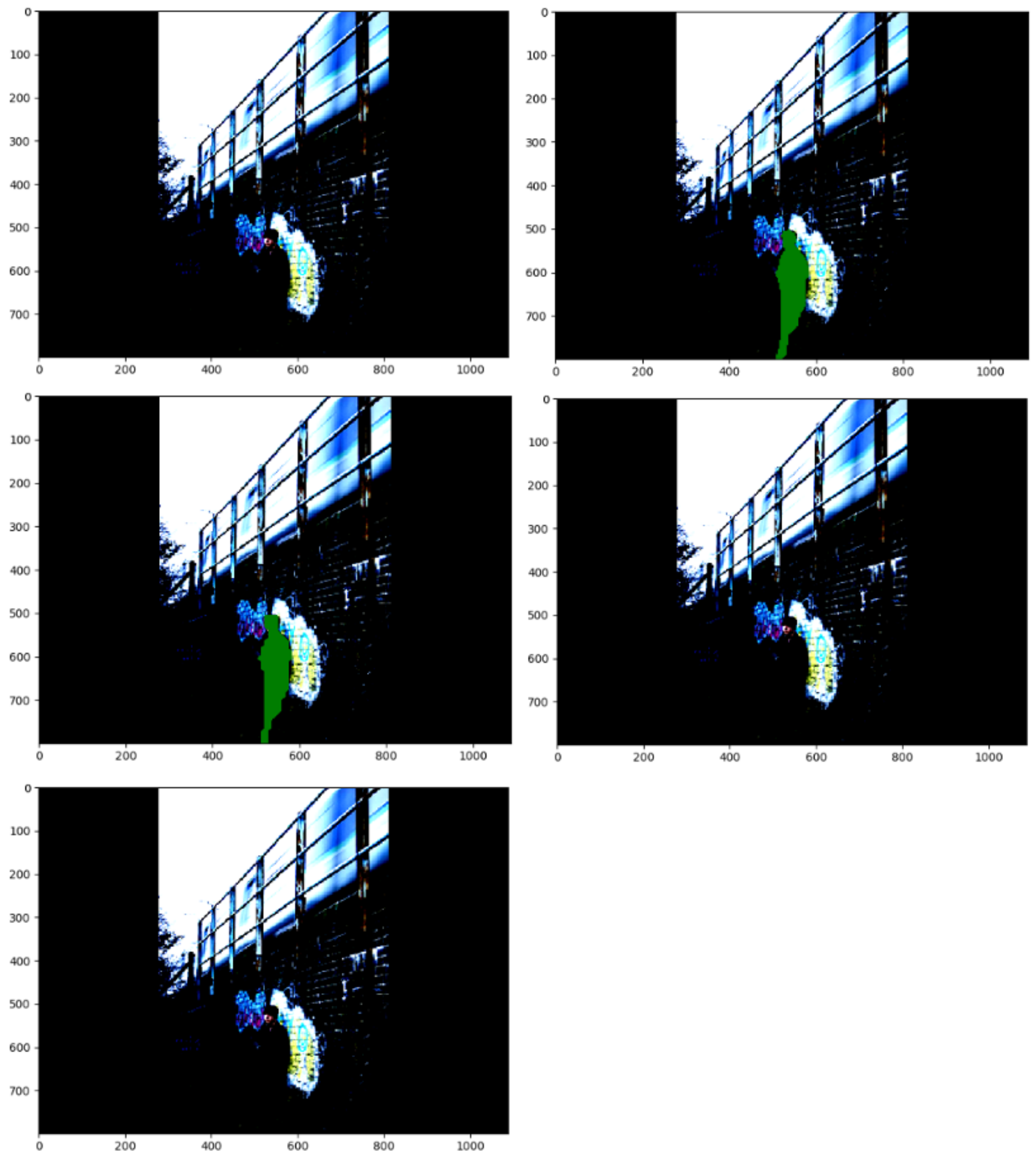


Figure 8: FPN mask visual example1

6.7 Submission Part-A

For the submission of Part-A.

- pdf report:

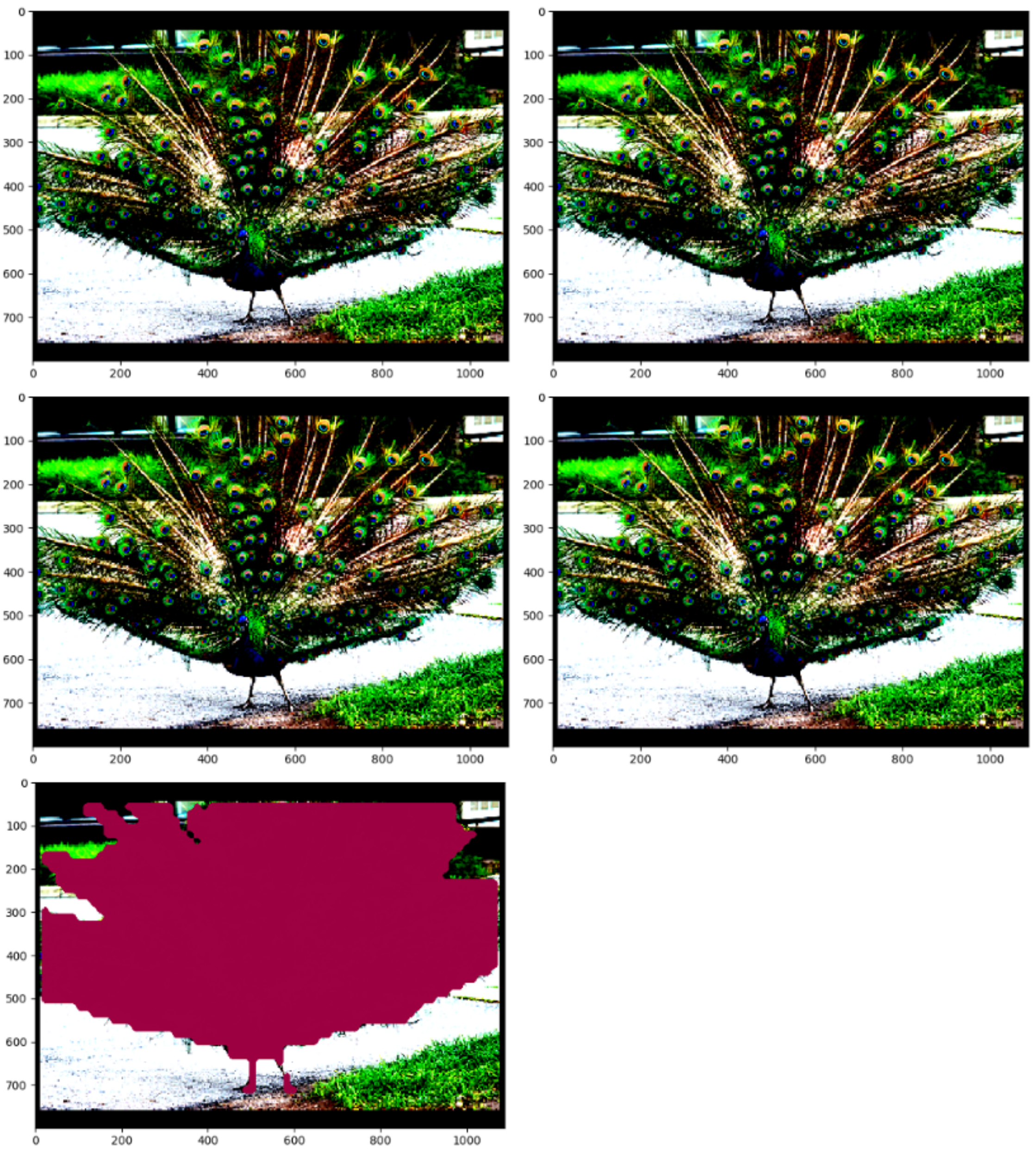


Figure 9: FPN mask visual example2

  - dataset plots as like Fig[5] and Fig[6]. At least include 5 examples and at least two of them include multiple objects.
  - FPN recover as like Fig[8] and Fig[9]. At least show 5 examples and at least two of them include multiple objects. Discuss what the difference of same object in different level of feature pyramid and why?
  - a README.md file including instruction how to run part-a codes.

- codes:
  - dataset.py should include an individual main function to plot dataset plots.
  - backbone.py
  - solo_head.py with part-a functions' implementation. Come with an individual main function to plot feature pyramid recovery.

6.8 Part - B

6.9 Loss

6.9.1 Loss function design

$$L = L_{\text{cate}} + \lambda L_{\text{mask}}$$

1. L_{cate} for category cost, L_{mask} for the segmentation cost, $\lambda = 3$.
2. $L_{cate} = \frac{1}{S^2 * C}\Sigma_{S,S,C}\text{FL}(p_t)$ based on focal loss which a positive and negative sample balanced cross entropy loss

   - $\text{FL}(p_\text{t}) = -\alpha_t (1-p_\text{t})^\gamma \log(p_\text{t})$, where $(\alpha_t, p_\text{t}) = \begin{cases} (\alpha, \hat{p}) & \text{if } y = 1 \\ (1-\alpha, 1-\hat{p}) & \text{o.w.} \end{cases}$
   - in the author's .cu code cross entropy is computed through every entry in the category output tensor i.e. $S \times S \times C$ entries

3. $L_{\text{mask}} = \frac{1}{N_{\text{positive}}} \sum_k \mathbb{I}_{\{\mathbf{p}_{i,j}>0\}} d_{\text{mask}}(\mathbf{m}_k, \mathbf{m}_k^*)$, where $d_{mask} = 1 - D(\mathbf{p}, \mathbf{q})$ is the Dice loss and $D(\mathbf{p}, \mathbf{q}) = \frac{2\Sigma_{x,y}(\mathbf{p}_{x,y} \cdot \mathbf{q}_{x,y})}{\Sigma_{x,y}\mathbf{p}_{x,y}^2 + \Sigma_{x,y}\mathbf{q}_{x,y}^2}$

6.9.2 loss

```
# This function compute loss for a batch of images
# input:
    # cate_pred_list: list, len(fpn_level), each (bz,C-1,S,S)
    # ins_pred_list: list, len(fpn_level), each (bz, S^2, 2H_feat, 2W_feat)
    # ins_gts_list: list, len(bz), list, len(fpn), (S^2, 2H_f, 2W_f)
    # ins_ind_gts_list: list, len(bz), list, len(fpn), (S^2,)
    # cate_gts_list: list, len(bz), list, len(fpn), (S, S), {1,2,3}
# output:
    # cate_loss, mask_loss, total_loss
```

Notice that here prediction's dimension order does not align with ground truth dimension. We use $zip()$ and $zip(*)$ function to manipulate dimension into desired shape. And use *ins_ind_gts_list* to pick out activated grids/channels.

The manipulation is quite difficult, codes is provided as follows:

```
## uniform the expression for ins_gts & ins_preds
# ins_gts: list, len(fpn), (active_across_batch, 2H_feat, 2W_feat)
# ins_preds: list, len(fpn), (active_across_batch, 2H_feat, 2W_feat)
ins_gts = [torch.cat([ins_labels_level_img[ins_ind_labels_level_img, ...]
                      for ins_labels_level_img, ins_ind_labels_level_img in
                      zip(ins_labels_level, ins_ind_labels_level)], 0)
```

```
           for ins_labels_level, ins_ind_labels_level in
           zip(zip(*ins_gts_list), zip(*ins_ind_gts_list))]

ins_preds = [torch.cat([ins_preds_level_img[ins_ind_labels_level_img, ...]
                        for ins_preds_level_img, ins_ind_labels_level_img in
                        zip(ins_preds_level, ins_ind_labels_level)], 0)
             for ins_preds_level, ins_ind_labels_level in
             zip(ins_pred_list, zip(*ins_ind_gts_list))]

## uniform the expression for cate_gts & cate_preds
# cate_gts: (bz*fpn*S^2,), img, fpn, grids
# cate_preds: (bz*fpn*S^2, C-1), ([img, fpn, grids], C-1)

cate_gts = [torch.cat([cate_gts_level_img.flatten()
                       for cate_gts_level_img in cate_gts_level])  # cate_gts_level_i
            for cate_gts_level in zip(*cate_gts_list)]
cate_gts = torch.cat(cate_gts)

cate_preds = [cate_pred_level.permute(0,2,3,1).reshape(-1, self.cate_out_channels)
              for cate_pred_level in cate_pred_list]
cate_preds = torch.cat(cate_preds, 0)
```

6.9.3 DiceLoss

```
# This function compute the DiceLoss
# Input:
    # mask_pred: (2H_feat, 2W_feat)
    # mask_gt: (2H_feat, 2W_feat)
# Output: dice_loss, scalar
```

6.9.4 FocalLoss

```
# This function compute the cate loss
# Input:
    # cate_preds: (num_entry, C-1)
    # cate_gts: (num_entry,)
# Output: focal_loss, scalar
```

6.10 PostProcess

6.10.1 PostProcess

```
# This function receive pred list from forward and do post-process
# Input:
    # ins_pred_list: list, len(fpn), (bz,S^2,Ori_H/4, Ori_W/4)
    # cate_pred_list: list, len(fpn), (bz,S,S,C-1)
    # ori_size: [ori_H, ori_W]
# Output:
    # NMS_sorted_scores_list, list, len(bz), (keep_instance,)
    # NMS_sorted_cate_label_list, list, len(bz), (keep_instance,)
    # NMS_sorted_ins_list, list, len(bz), (keep_instance, ori_H, ori_W)
```

6.10.2 PostProcessImg

```
# This function Postprocess on single img
# Input:
    # ins_pred_img: (all_level_S^2, ori_H/4, ori_W/4)
    # cate_pred_img: (all_level_S^2, C-1)
# Output:
    # NMS_sorted_scores_list, list, len(bz), (keep_instance,)
    # NMS_sorted_cate_label_list, list, len(bz), (keep_instance,)
    # NMS_sorted_ins_list, list, len(bz), (keep_instance, ori_H, ori_W)
```

$all\_level\_S^2$ means add up S^2 at all levels. i.e. $40^2+36^2+24^2+16^2+12^2$. Consider here you are gathering all predictions from different FPN levels.

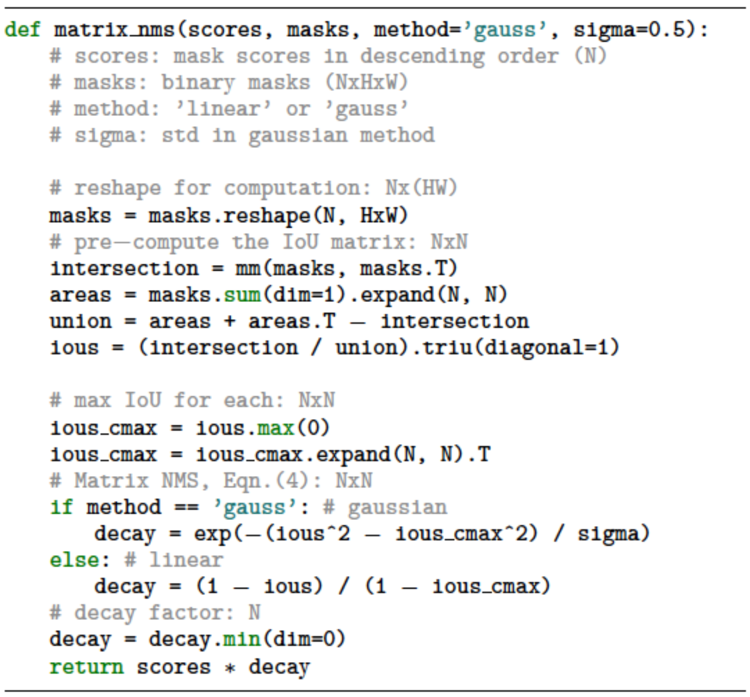
6.10.3 MartixNMS

```
# This function perform matrix NMS
# Input:
    # sorted_ins: (n_act, ori_H/4, ori_W/4)
    # sorted_scores: (n_act,)
# Output:
    # decay_scores: (n_act,)
```

Remember to make mask hard for sorted_ins, that is if pixel is mask, its value is 1.
inspired by soft-NMS method which compute the decay factor and suppress the score recursively until it below the shreshold.

- $IoU_{i,j} = flat(m_i) \cdot flat(m_j)^T$, m_j is of shape (H_I, W_I)
- where s_j is the score of j-th mask. $s_j = \frac{\Sigma_{\{y,x\}} \hat{m}_{j_{\{y,x\}}} \cdot \mathbb{I}_{\hat{m}_{j_{\{y,x\}}} > m_{thresh}}}{\Sigma_{\{y,x\}} \mathbb{I}_{\hat{m}_{j_{\{y,x\}}} > m_{thresh}}} \cdot c_j$ (found in the code)
- $f(IoU_{\cdot,i}) = \min_{\forall s_k > s_i} f\left(IoU_{k,i}\right)$
- $\text{decay}_j = \min_{\forall s_i > s_j} \frac{f\left(IoU_{i,j}\right)}{f\left(IoU_{\cdot,i}\right)}$
- compute a decay factor for each mask and update $s_j = s_j * decay_j$
- Keep first k masks according to the score.

MatrixNMS pseudocode refer to Fig.[10]
And here is an illustracate example of how it works:



```
def matrix_nms(scores, masks, method='gauss', sigma=0.5):
    # scores: mask scores in descending order (N)
    # masks: binary masks (NxHxW)
    # method: 'linear' or 'gauss'
    # sigma: std in gaussian method

    # reshape for computation: Nx(HW)
    masks = masks.reshape(N, HxW)
    # pre-compute the IoU matrix: NxN
    intersection = mm(masks, masks.T)
    areas = masks.sum(dim=1).expand(N, N)
    union = areas + areas.T - intersection
    ious = (intersection / union).triu(diagonal=1)

    # max IoU for each: NxN
    ious_cmax = ious.max(0)
    ious_cmax = ious_cmax.expand(N, N).T
    # Matrix NMS, Eqn.(4): NxN
    if method == 'gauss': # gaussian
        decay = exp(-(ious^2 - ious_cmax^2) / sigma)
    else: # linear
        decay = (1 - ious) / (1 - ious_cmax)
    # decay factor: N
    decay = decay.min(dim=0)
    return scores * decay
```

Figure 10: MatrixNMS pseudocode

we have 3 predictions with scores: $[0.9, 0.8, 0.8]$

1	1	0
1	1	0
0	0	1

0	0	1
0	0	1
0	0	0

1	1	0
1	1	0
0	0	0

Then we get the following matrix results:

$$masks = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}; \; intersections = \begin{bmatrix} 5 & 0 & 4 \\ 0 & 2 & 0 \\ 4 & 0 & 4 \end{bmatrix}; \; areas = \begin{bmatrix} 5 & 5 & 5 \\ 2 & 2 & 2 \\ 4 & 4 & 4 \end{bmatrix};$$

$$union = \begin{bmatrix} 5 & 7 & 5 \\ 7 & 2 & 6 \\ 5 & 6 & 4 \end{bmatrix}; ious = \begin{bmatrix} 0 & 0 & 4/5 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}; ious\_cmax = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 4/5 & 4/5 & 4/5 \end{bmatrix}; decay\_mat = \begin{bmatrix} 1 & 1 & 1/5 \\ 1 & 1 & 1 \\ 5 & 5 & 5 \end{bmatrix};$$

$decay = [1, 1/5]$.

That is to say, the score of 3-rd prediction is decreased to 1/5. While the first and second prediction score remain the same. This makes sense, since we observe that 1-st prediction and 3-rd prediction is highly overlapped and 1-st prediction has higher confidence. While 2-nd prediction have no intersection with others.

6.10.4 points_nms

The functionality of point NMS is process a 2d max pooling of kernel size 2 to the category branch heatmap output, to eliminate the neighboring grid having the same prediction for object. Notice that this 2d max pooling is applied to each channels independently.

```
# Credit to SOLO Author's code
# This function do a NMS on the heat map(cate_pred), grid-level
# Input:
    # heat: (bz,C-1, S, S)
# Output:
    # (bz,C-1, S, S)
```

This session code will be provided as follows:

```
def points_nms(self, heat, kernel=2):
    # kernel must be 2
    hmax = nn.functional.max_pool2d(
        heat, (kernel, kernel), stride=1, padding=1)
    keep = (hmax[:, :, :-1, :-1] == heat).float()
    return heat * keep
```

points_nms is used in inference,

```
cate_pred = self.points_nms(cate_pred).permute(0,2,3,1)
```

6.11 PlotInfer

```
# This function plot the inference segmentation in img
# Input:
    # NMS_sorted_scores_list, list, len(bz), (keep_instance,)
    # NMS_sorted_cate_label_list, list, len(bz), (keep_instance,)
    # NMS_sorted_ins_list, list, len(bz), (keep_instance, ori_H, ori_W)
    # color_list: ["jet", "ocean", "Spectral"]
    # img: (bz, 3, ori_H, ori_W)
```

For mask prediction, we set a hard threshold, that is,

```
mask[mask >= 0.5] = 1
mask[mask < 0.5] = 0
```

Set a threshold for the NMS score to select instance segmentation. Here are some example images(the color is used for category, you could also use text and random color) Fig.[11][12][13][14]:

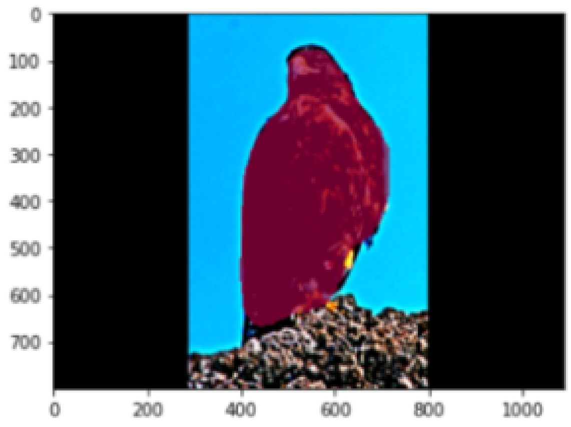


Figure 11: SOLO instance segmentation example 1

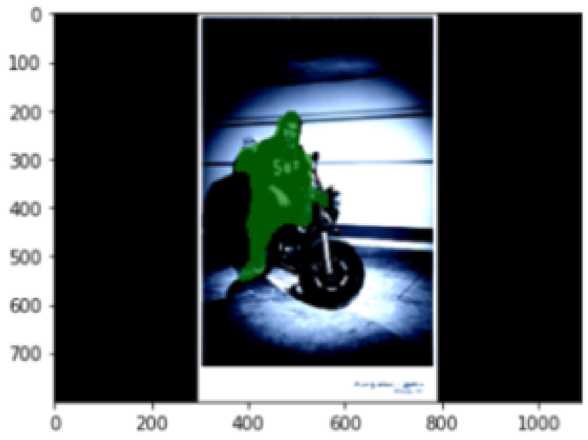


Figure 12: SOLO instance segmentation example 2

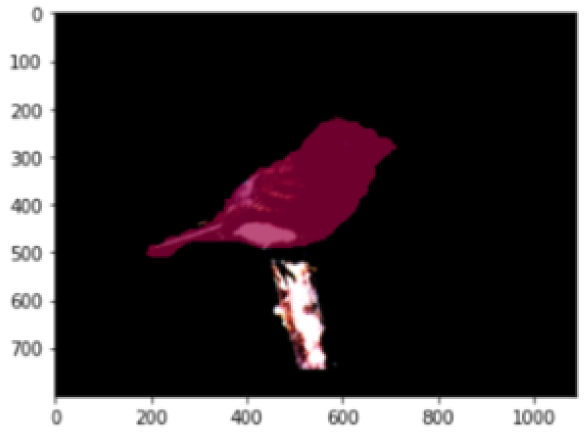


Figure 13: SOLO instance segmentation example 3

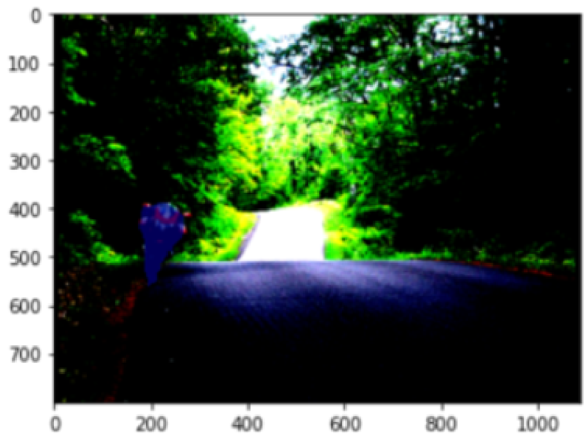


Figure 14: SOLO instance segmentation example 4

7 main_train.py

7.1 optimizer

SGD optimizer with weight decay of 0.0001 and momentum of 0.9 are used. Initial learning rate is 0.01 for a batch size of 16 images. Initial learning rate are changed w.r.t. batch size. If you are using a batch size of 2 images, use 0.01/8 as initial learning rate. Total training epoch is 36, divide learning rate by 10 at 27-th and 33-th epoch.

7.2 data augmentation

you could apply data augmentation as you like, remember to include the data augmentation you used in the final report. The example prediction result shown above is learning without data augmentation.

7.3 training checkpoint

You need GPU to speed the training, you could use colab to access the GPU. One tip is that you may want to save checkpoint of model after each epoch. Take a look here for how to save checkpoint:
https://pytorch.org/tutorials/recipes/recipes/saving_and_loading_a_general_checkpoint.html

8 main_infer.py

In inference, set model to eval model by passing eval=True to the solo_head class. Remember we have different behavior in forward with different eval mode.

9 Submission Part-B

PDF Write-up submission

1. Final instance segmentation prediction plot like Fig.[11][12][13][14]. Include at least 5 results, at least one include multiple objects at the same scene. And at least one contain multiple category objects.

2. Include losses curve w.r.t. iterations, you could set log numbers as 100 mini-batch or a reasonable number. Plot should include $total\_loss$, $focal\_loss$, and $Dice\_loss$.

3. You are free to change the parameters setting, but include all the settings in the report.

4. Choose a reasonable confidence threshold and compute mAP value and Precision-recall curve for each category. Notice this is very similar to YOLO project, only here IoU is mask IoU(covered in Matrix NMS) other than the Bounding Box IoU.

5. (optional) report the data augmentation if any.

Code submission

1. A .zip file includes all the codes. Here is a checklist:

   - *dataset.py*,
   - *backbone.py*,
   - *solo_head.py*,
   - *main_train.py* / *resume_train.py*,
   - *main_infer.py*.

   You could use your own structure, but make clear the function of each .py file and include in the README.md file.

2. And a README.md file including instruction of how to run the code.

This is a very challenging project, so we encourage you to start as early as possible. Please submit your completed work to Gradescope. The submission should include a pdf of your report and a zip file with only the Python files for your implementation.

10 Some Useful Online Materials

Original SOLO paper:
https://arxiv.org/pdf/1912.04488.pdf
Introduction PDF:
https://drive.google.com/file/d/1hjhkwr5B9idJyg2wqj$_{p}yYx4ytDBSTp6/view?usp = sharing$
mean Average Precision:
mean Average Precision Tutorial