SOLO IN BIOMEDICAL IMAGING : AN EXPLORATION

REPORT

Hanwen Wang
Graduate Group of Applied Mathematics
and Computational Science
University of Pennsylvania
Philadelphia, PA 19104
wangh19@sas.upenn.edu

Yifei Li
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104
liyifei@seas.upenn.edu

Zhenglin Zhang
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104
zlzhang@seas.upenn.edu

January 23, 2022

ABSTRACT

In this project, we will explore the combination of traditional biomedical imaging dataset with state-of-the-art instance segmentation model. From moderated compressed dataset created with commercial software, we first use DBSCAN algorithm to convert them to standard data format that is compatible to the usage of data in computer vision field. We than modified the SOLO with a customized ResNet50 backbone for a binary classification problem, and searched a few hyperparameters to tune the model from COCO dataset to biomedical dataset. The average precision of the four sets of hyperparameters are 0.473, 0.584, 0.647 and 0.461. We also make a few attempts to add GRU for FPN output levels, but due to the extra complexity that it brings, none of them succeeds. A brief video presentation of the project can be found in the link included in the footnote[1].

Keywords Hemostatic Plug · Biomedical Imaging · Computer Vision · Instance Segmentation

Introduction

Hemostatic plug, also known as the platelet plug, is a aggregation of platelets forming around injuries of the blood vessel wall as a part of the hemostasis mechanism. The platelets in the blood scream are activated by the exposure to the non-blood vessel environment, whose adhesive building up forms a preliminary protection against external contaminants as well as further blood loss [1]. However, pathological formation of hemostatic plugs is often a crucial part of fatal diseases. For example, in hemophilia and von Willebrand disease, the patients' platelets fail to produce fiberin that is essential to the stabilization of the plug. As a consequence, the plug breaks down and exposes the injuries[2]. Such mechanism brings the need of early diagnosis of pathological formations of the plug from patterns of its fiberin backbone.

While the current studies of hemostatic plug formation rely heavily on manually annotated platelets on the scanning of the slices of the plug (as shown in figure 1), in this project, we explore an automatic annotation and separation pipeline for sliced scanning of the plug using the SOLO[3] without the burden of traditional segmentation algorithms that cannot be implemented in parallel fashion. The one-shot prediction of individual cell masks also allows the incorporation of

[1] https://drive.google.com/file/d/1VhEnA24MkdNraOXgZryIU_ZDg8BbFp6J/view?usp=sharing

spatial prior information, where the channels of images are continuous scanning of slices of tissue, or the evolution of the cells in time scale, that leads to the 3-D point clouds reconstruction of individual cells with an additional $z-$axis.

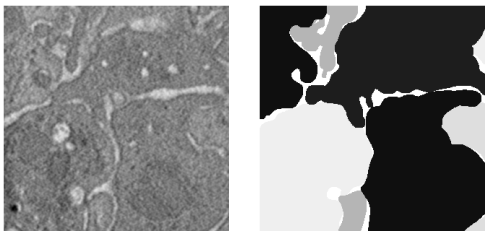


Figure 1: A visualization of existing data. (Left) Original scanning. (Right) Manually annotated mask.

Related Work Echoing with the vibrant community of deep learning, the computer vision researchers are presenting an unprecedentedly rapid development of instance segmentation methods that efficiently detect the presence of objects from multiple categories in images without human intervention. While achieving considerable success in Common Object in Context (COCO) dataset[4], the state of the art instance segmentation models, such as SOLO [3], SSD [5], and Mask R-CNN [6], are rarely seen in the fields of biomedical imaging, where U-Net based methods dominate especially for cell detection and identification in the microscopic tissue images. On the contrary to the popular instance segmentation models mentioned above, where the masks are predicted after the determination of their locations, these U-Net based methods predict semantic segmentation mask first for the entire image, before using algorithms such as watershed or regional proposal network [7] [8] [9] to identify masks for individual objects in the image. The inversed two-stage detection fails to anchor the masks at their locations and therefore compromising the ability to trace individual objects across the images.

Meanwhile, region-based Convolutional Network method (R-CNN) [10] and its descendants, e.g. Fast R-CNN [11] and Faster R-CNN [12], contribute a popular diagram for object detection. They use regional proposal or bounding box to localize and segment the target. Following the work of Faster R-CNN, Mask R-CNN [6] extends this method to instance segmentation task by adding an extra branch of mask prediction besides the bounding box prediction branch as an end-to-end solution. It's easy to be trained and regularized. Later, Wang, X. *et al.* [3] introduced Segmenting Objects by Locations (SOLO). By assigning categories on pixel level via the instance's spatial information (location and size), it successfully transforms the traditional instance segmentation to a one-shot classification-solvable task, achieving on par performance with Mask R-CNN but with a simpler one-shot architecture.

Methodology

This section describes the pre-processing, inference architecture, model training, and post-processing, and evaluation metrics in details. All the codes can be found in the supplemental material appended in the end.

Dataset description As shown in 1, the dataset comprises 69 original scanning of the hemostatic plug with size 1250×1250, and manually annotated mask where the area of the platelets are labeled with different grey scales or RGB colors. We first crop each images to 16 images with size 313×313 to reduce the size and increase the number of sample. Among these 1104 images, we use $80 - 20$ split for training set and testing set. We are requested by the data providers to not disclose the dataset.

Dataset pre-processing To overcome such anomaly from compression and extract individual masks from the manually labeled mask for entire images, we use Density-based spatial clustering of applications with noise (DBSCAN)[13] to identify clusters of pixels (i.e., individual platelet masks) with close grey scale or RGB color values. DBSCAN classifies points as core points, boundary points, and noise points from the number of in their ϵ-neighborhood. Then core points that are close to each others, as well as boundary points that are close to the core points of same labels, are identified as a cluster. Specifically, we use the Euclidean distance measured together on the positional indices and color space as a distance metric for the pixels. We use $\epsilon = 2$ to define neighborhood of a point, and $N = 9$ to define how many points in neighborhood that a point must have in order to be a core point. Figure 2 shows a sample extraction of

individual masks.

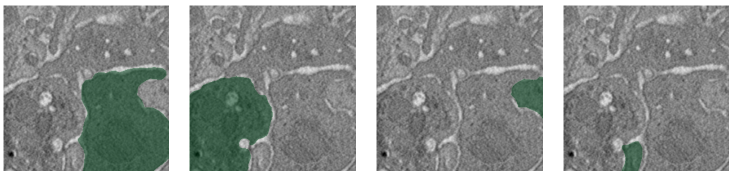


Figure 2: Examples of individually extracted mask.

Once the masks are obtained, the bounding box parameters are chosen as 0.1% to 99.9% quantiles at each dimension to avoid lingering pixels.

For target assignment to FPN feature levels, we follow the routine in the original SOLO paper, and test a variety of combination of scale ranges and feature grid sizes. However, since the we have limited knowledge of the dataset, our choices of these hyperparameters may not be optimal.

FPN backbone We use Resnet50 + 4 layers Feature Pyramid Network as the backbone of SOLO. The difference we made about the regular Resnet is that we chose 4 FPN layers so that the first feature map produced by the first residual block should also be taken into consideration and parameters need to be adjusted. The detail of the process is similar to regular FPN: each layer comes after the residual block, after upsampling and soothing then blend with the lower layer, the final feature maps of each layer are used to predict different sizes of masks.

Recurrent structure To utilize the spatial information provided in the sliced scanning, we use a recurrent structure that combines features from nearby slices. Among sequence processing models, Gated Recurrent Units (GRU) [14] has been a popular light-weighted alternatives to the older Long Short-Term Memory (LSTM)[15]. Empirically, it has better performance on certain smaller and rare datasets. In 2015, Bi-LSTM [16] was proposed by Huang *et al.* as a bidirectional variant of LSTM that utilizes both the past and future information. We will similarly construct a bidirectional variant of GRU, where the linear layers are replaced by convolutional layer with kernel size 3×3. Instead of doubling the hidden dimension, we first concatenate the hidden state from both direction together, and then apply a 1×1 convolution to recover the original dimension. Figrue 3 shows a schematic drawing of the architecture with recurrent units.

SOLO head As mentioned in the related work, we implemented SOLO for instance detection, basically following the original architecture setting from the paper. However, as a binary classification task, the mask loss function is simplified from focal loss to binary cross entropy.

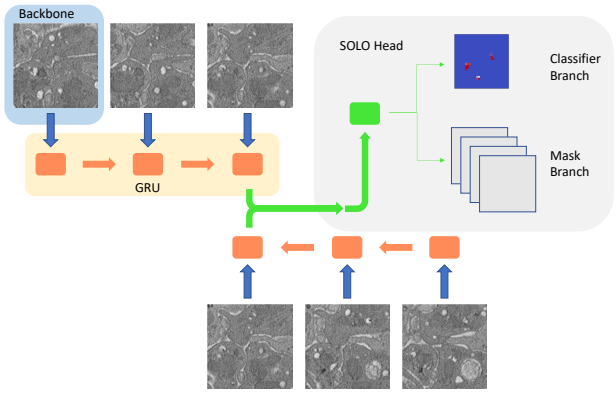


Figure 3: Architecture of SOLO with GRU.

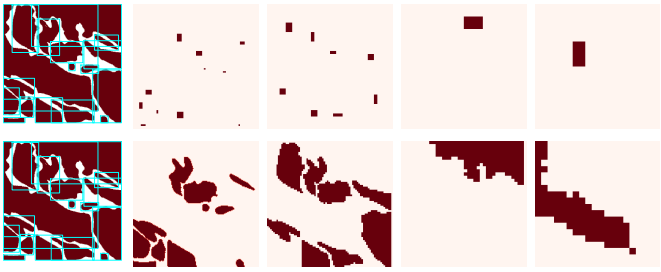


Figure 4: A visualization of target assignments. On the left is a picture of masks bounded by their bounding boxes. (Top) Activated pixels on the feature map in each layer. (Bottom) The corresponding masks assigned to the activated pixels.

Post-processing For each image, we choose top 200 scored masks whose categorical score exceeds 0.5. Then their confidence score, a multiplication of the categorical and maskness, are used for the matrix Non-Maximal Suppression (NMS), after which top 15 masks are output as final prediction.

Evaluation metrics Mean Average Precision (mAP) is a popular metrics in object detection domain. By computing the average precision of each class detection with the help of IoU threshold and then averaging it by the number of class, we can know how well the model performs.

Training We choose Adam optimizer [17] for its fast convergence. Although heuristic circulates that the SGD with momentum may promote better generalization, our pursuit to higher test performance was hindered by the limited time. We choose the learning rate as $1e-3$ as standard practice, and have annealing schedule per 15 epoch by a factor of 0.95. The loss function is designed as

$$L = \lambda_c L_{cate} + \lambda_m L_{mask}. \tag{0.1}$$

L_{cate} is the binary cross entropy loss for the classifer head that distinguish a feature grid from background, and L_{mask} is the mask loss that can be expressed as

$$L_{mask} = \frac{1}{N_{pos}} \sum d_{mask}(p^i, q^i), \tag{0.2}$$

where p^i, q^i are the assigned mask, and the predicted mask, respectively. And d_{mask}, the dice loss, is defined as

$$d_{mask} = 1 - \frac{2\langle p, q\rangle}{\langle p, p\rangle + \langle q, q\rangle}, \tag{0.3}$$

where $\langle \cdot, \cdot \rangle$ is the inner product on the flattened masks.
We use $\lambda_c = \lambda_m = 1$ as the weight. Figure 5 shows the descending of loss curves during the training for a successful

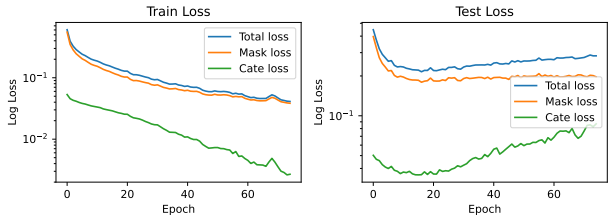


Figure 5: Loss curves of a successful training of SOLO.

trial. Although the testing classification loss increases, we consider that the overfitting is also a necessary part for the reconstruction of training set, and we therefore continue training. The test mask loss stays almost constant during the training.

Result

Inference result Figure 6 shows sample outputs from train and test images. Due to the page limit, we cannot provide an exhausting list of post-processed prediction.

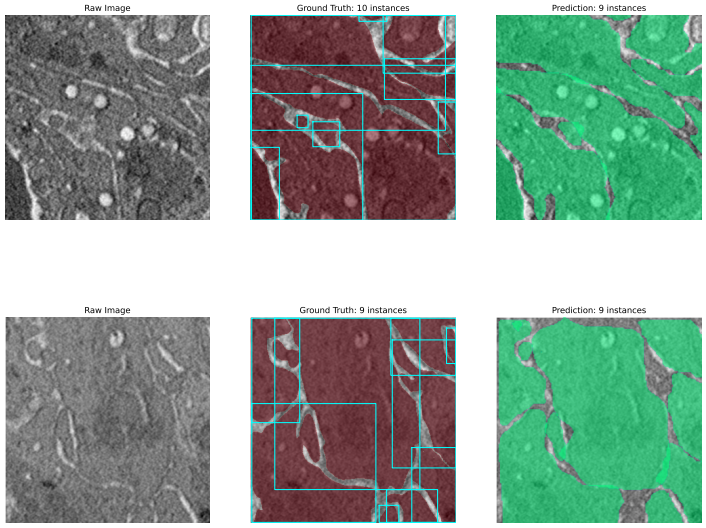


Figure 6: Sample post processed inference results from SOLO without GRU. (Top) On one of the train images. (Bottom) On one of the test images.

For SOLO with GRU, unfortunately, the model fails to learn and does not detect any object due to the extremely low confidence score returned by the model. Both the implementation given by earlier studies and our own fail for the same reason. Figure 7 shows the loss curves during the training, that fail to decrease. We will therefore not provide any result for SOLO with GRU, but we will include a section for the potential cause and fix for the failure in the discussion section later.

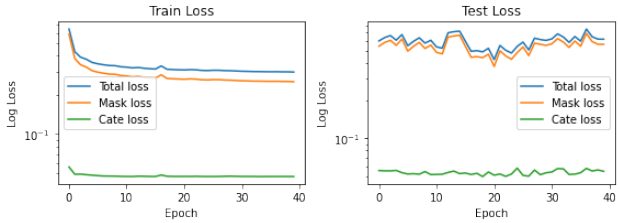


Figure 7: Loss curves of SOLO with GRU fail to decrease.

Average percision For each architecture, we use 4 fixed sets of scale ranges for target generation to test their performance (Table 1). Among them, Model 3 always gives the highest mAP.

Discussion and conclusion

As expected, unlike the model on the more mundane COCO dataset, the SOLO head requires more concrete and diverse levels of features to cope with the densely packed masks across the entire image, whereas in COCO, the targets are usually much more scattered. However, even without optimal hyperparameters, the model returns visually acceptable results except at some extreme case caused by the cropping of images. Overall the performance of the model is above

	Model1†	Model2†	Model3‡	Model4§
mAP (all mask)	0.473	0.584	0.647	0.461
mAP (mask size>25)	0.594	0.676	**0.769**	0.635
mAP (mask size>50)	0.468	0.617	0.603	0.597

Table 1: Mean Average Precision of Different Models on the test set. Note that we also calculate average precision discarding overly small masks from the truncation error of the original data.
†,†,‡,§, Grid number as [60,40,20,20], [80,40,20,10], [80,60,30,10] and [90,60,30,10] respectively.
†,†,‡,§, Scale range as (0-.4, .2-.6, .4-.8, .6-1), (0-.25, .2-.5, .4-.8, .6-1), (0-.3, .1-.4, .3-.7, .6-1) and (.05-.4, .1-.5, .4-.7, .6-1) respectively.

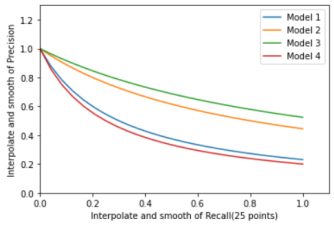


Figure 8: Precision-Recall curves of mAP of four model settings (details are in the Table 1's caption) tested evaluated on all-mask test set. It's smoothed by 25-point nonlinear interpolation for better visualization.

expectation, since the cells in the sliced scanning is much harder even for human to recognize, as compared to the COCO dataset where the objects are often more familiar to human. We believe that adaption of vision model for common object could be very helpful for the field of biomedical imaging.

Comment on SOLO with GRU As mentioned above, SOLO with GRU fails to learn. However, this result is still within reasonable expectation since the field convolutional recurrent neural network in general has only limited studies. While the convolution itself is an analog of linear operation on a regular grid instead of a vector, whether we can replace the linear operation in recurrent structure with convolution is still unknown. And in literature[18], researchers also admitted that the introduction of recurrent structure may not always bring at least comparable result.

Besides the architecture, the errors in manual annotation also accumulate and bring conflicts to scanning of even nearby slices, due to the fact that human is not too much capable of recognizing spatial correlation, especially when the images to be labeled are not fed with spatial order. Such discrepancy in the images data makes the introduction of GRU as well as additional complexity far less economic. However, we do believe that based on the intuition, that larger cells are more likely to have significant spatial correlation, we can apply the recurrent architectures to the last few FPN layers which are assigned to detect those larger objects.

Future work

Limited by time, we directly use most of the hyperparameters from the original SOLO paper, which reportedly have high performance on the COCO dataset. However, the discrepancy between the COCO dataset and the sliced scanning of hemostatic plug indeed plays a roll in the relatively low performance that our model has. Fine tuning for hyperparameters, such as ϵ defining the center region, target assignment scale ranges, as well as feature grid sizes, may be the top priority for the future work. Architecture-wise, we may also reconsider a more appropriate FPN structure that involves U-net whose hyper-resolution capability has been widely recognized in the field of biomedical application. Besides architectures and training procedures, we may continue working on 3-D object reconstruction from the 2-D masks predicted in one location. Ultimately we can apply NMS to the 3-D mask point clouds and return accurate binary reconstruction of point clouds of the hemostatic plug and use it a a base for cellular dynamic simulation.

Authors' contributions

Hanwen Wang perceived the methodology, designed the architectures, implemented the gated recurrent unit, and prepared the data. Yifei Li implemented the SOLO head, and the training and validation pipeline. Zhenglin Zhang implemented the Feature Pyramid Network from ResNet50 backbone and evaluation of the model result. Zhenglin Zhang and Yifei Li designed and implemented the post-processing metric. The three authors collectively wrote the manuscript.

Acknowledgement

We thank Prof. Stalker from Jefferson University Dr. Tomaiuolo from Penn Medicine, and Prof. Sinno and Prof. Perdikaris from Penn Institue of Computational Science (PICS) for generously allowing us to use their unpublished data as well as providing incomparable computational resource. We also thank the course instructor Prof. Shi and the teaching assistance Lukas for their support and advice on data preprocessing and methodology.

References

[1] J. Ware and Z.M. Ruggeri. Platelet adhesion receptors and their participation in hemostasis and thrombosis. *Drugs of Today*, 37(4):265, 2001. ISSN 1699-3993. doi:10.1358/dot.2001.37.4.620592. URL `http://journals.prous.com/journals/servlet/xmlxsl/pk_journals.xml_summary_pr?p_JournalId=4&p_RefId=620592&p_IsPs=N`.

[2] Northwestern Medicine. Blood Clot Disorder. URL `https://www.nm.org/conditions-and-care-areas/hematology/blood-clot-disorder`.

[3] Xinlong Wang, Tao Kong, Chunhua Shen, Yuning Jiang, and Lei Li. Solo: Segmenting objects by locations. In *European Conference on Computer Vision*, pages 649–665. Springer, 2020.

[4] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2015.

[5] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[6] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[7] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[8] Zongwei Zhou, Md Mahfuzur Rahman Siddiquee, Nima Tajbakhsh, and Jianming Liang. Unet++: A nested u-net architecture for medical image segmentation. In *Deep learning in medical image analysis and multimodal learning for clinical decision support*, pages 3–11. Springer, 2018.

[9] Ozan Oktay, Jo Schlemper, Loic Le Folgoc, Matthew Lee, Mattias Heinrich, Kazunari Misawa, Kensaku Mori, Steven McDonagh, Nils Y Hammerla, Bernhard Kainz, et al. Attention u-net: Learning where to look for the pancreas. *arXiv preprint arXiv:1804.03999*, 2018.

[10] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Region-based convolutional networks for accurate object detection and segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 38 (1):142–158, 2015.

[11] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[12] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 201, 2015.

[13] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press, 1996.

[14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997. ISSN 0899-7667. doi:10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

[16] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*, 2015.

[17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[18] Martin Zihlmann, Dmytro Perekrestenko, and Michael Tschannen. Convolutional recurrent neural networks for electrocardiogram classification. In *2017 Computing in Cardiology (CinC)*, pages 1–4, 2017. doi:10.22489/CinC.2017.070-060.

Supplemental Material

Computational cost All the computations are run on a single NVIDIA RTX A6000 graphic card with 48 Gb memory. Table 2 shows some general computational cost of the chosen model.

Model	Evaluation	Post-Inference	Time per epoch
SOLO	44 ± 8.52 ms	197 ± 27.8 ms	~ 160 s
SOLO with GRU	133 ± 15 ms	508 ± 142 ms	~ 320 s

Table 2: Computation cost of evaluation (a batch of 4), inference with post processing (single image), and a training epoch (~ 820 images). All the computations are run on a single NVIDIA RTX A6000 graphic card with 48 Gb memory.

Codes

We provide a comprehensive list of the codes we have updated from the earlier submission. The saved model weights for this project will be saved for at least a month, and is available upon request.

Pre Processing

dbscan_bw

December 17, 2021

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from mpl_toolkits.mplot3d import Axes3D
from sklearn.cluster import DBSCAN
from tqdm import tqdm, trange
import os
```

[3]:
```
from torch import nn
import torch
```

[4]:
```
path = '../processdata'
```

[5]:
```
mask_img = Image.open(f"{path}/sublabeled/0-0-0-0.tif")
image = np.array(Image.open(f"{path}/suboriginals/0-0-0-0.tif"))
```

[6]:
```
mask = np.array(mask_img)
```

[7]:
```
plt.figure(figsize = (5,5))
plt.imshow(mask, cmap = 'gray')
plt.axis('off')
plt.show()
plt.close()
```

[8]:
```
offset = 0.5
def estimate_class(mask):
    # use the average of the grayscale to determine class
    return np.mean(mask[np.where(mask > 0.5)]), np.std(mask[np.where(mask > 0.
↪5)])
```

[9]:
```
mask_coord = np.array(np.meshgrid(np.arange(mask.shape[0]), np.arange(mask.
↪shape[1])))
mask_val = mask * offset

mask_data = np.stack((*mask_coord, mask_val), 0).copy()
flat_points = mask_data.reshape(3,-1).T

normalized_coords = mask_coord / np.max(mask_coord, axis = (1,2)).reshape(2,1,1)
```

[10]:
```
# eps for color distortion
# DBSCAN from sklearn to extract individual masks.
clustering = DBSCAN(eps = 2, min_samples = 9).fit(flat_points)
labels = clustering.labels_
```

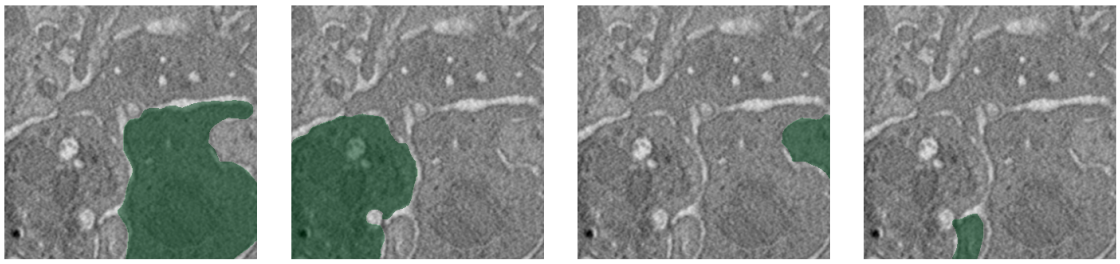
[11]:
```
all_mask = []

for i in tqdm(np.unique(labels)[1:]):
```

```
    if flat_points[np.where(labels == i)[0],-1].mean() / offset < 245:
        all_mask.append(np.where(labels == i, flat_points[:,-1] / offset, 0).
 ↪reshape(*mask.shape[:2]))
all_mask = np.array(all_mask)
```

```
100%|      | 31/31 [00:00<00:00, 3984.66it/s]
```

[12]:
```
# Remove noise points, boundary points and overly small masks.
all_class = np.array(list(map(estimate_class, all_mask)))

valid_class = all_class[np.where(all_class[:,0] < 245)[0]]
valid_mask = all_mask[np.where(all_class[:,0] < 245)[0]]

valid_mask = valid_mask[valid_class[:,1] < 3]
valid_class = valid_class[valid_class[:,1] < 3]
px_thresh = 10
valid_class = valid_class[np.sum((valid_mask > 0.5).astype(int), axis = (1,2))␣
 ↪> px_thresh]
valid_mask = valid_mask[np.sum((valid_mask > 0.5).astype(int), axis = (1,2)) >␣
 ↪px_thresh]

valid_mask = (valid_mask > 0.5).astype(int) * valid_class[:,0][:,None, None].
 ↪astype(int)
```

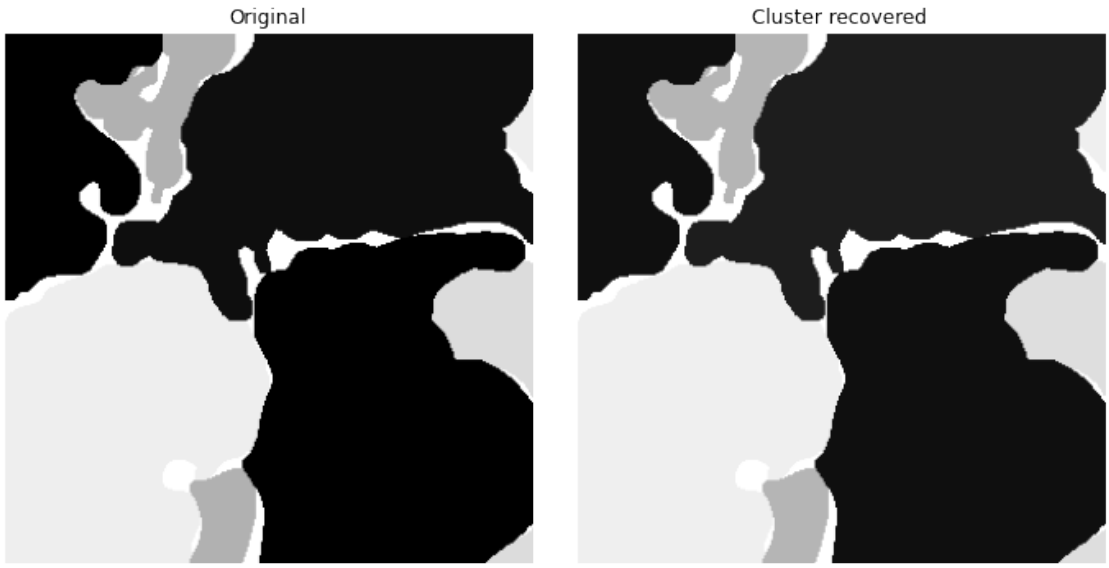
[13]:
```
count = 0
plt.figure(figsize = (4 * 5, 5))

for m, c in zip(valid_mask[4:8], valid_class[4:8]):

    count += 1

    plt.subplot(1, 4, count)
    plt.imshow(image)
    plt.imshow((m > 0).astype(int), cmap = 'Greens', alpha = (m > 0).
 ↪astype(int) * 0.5)
    plt.tight_layout()

    plt.axis('off')

plt.savefig('all_mask_example.pdf', dpi = 100)
plt.show()
plt.close()
```



[14]:
```
count = 0
plt.figure(figsize = (4 * 5, (valid_mask.shape[0] // 4 + 1) * 5))

for m, c in zip(valid_mask, valid_class):

    count += 1

    plt.subplot((valid_mask.shape[0] // 4 + 1), 4, count)
    plt.imshow((m > 0).astype(int), cmap = 'Reds')
    plt.title(f'Mean : {c[0]:.2f}, var : {c[1]:.2f}')

    plt.axis('off')

# plt.savefig('all_mask.pdf', dpi = 100)
plt.show()
plt.close()
```

[26]:
```
mask_cumulative_sum = np.where(valid_mask.sum(0).astype(int) == 0, 255,␣
↪valid_mask.sum(0))


plt.figure(figsize = (10,5))


plt.subplot(1,2,1)
plt.imshow(mask, cmap = 'gray')
plt.title('Original')
plt.axis('off')
plt.tight_layout()

plt.subplot(1,2,2)
plt.imshow(mask_cumulative_sum, cmap = 'gray', vmin=0, vmax=255)
plt.title('Cluster recovered')
plt.axis('off')
plt.tight_layout()

plt.savefig('recovered.pdf', dpi = 100)
```

```
plt.show()
plt.close()
```



[ ]:
```
q = np.array([0.001, 0.999])
for t in trange(50):
    for x in range(4):
        for y in range(4):
            file_name = f"0-{x}-{y}-{t}"

            mask_file_name = path + "submasks/" + file_name + ".npy"
            bbox_file_name = path + "subbboxes/" + file_name + ".npy"

#             if not (os.path.isfile(mask_file_name) and os.path.
 ↪isfile(bbox_file_name)):
            if True:

                mask_img = Image.open(path + "sublabeled/" + file_name + ".tif")

                mask = np.array(mask_img)

                mask_coord = np.array(np.meshgrid(np.arange(mask.shape[0]), np.
 ↪arange(mask.shape[1])))
                mask_val = mask * offset

                mask_data = np.stack((*mask_coord, mask_val), 0).copy()
                flat_points = mask_data.reshape(3,-1).T
```

```
                normalized_coords = mask_coord / np.max(mask_coord, axis =␣
↪(1,2)).reshape(2,1,1)

                mask_coord = np.array(np.meshgrid(np.arange(mask.shape[0]), np.
↪arange(mask.shape[1])))
                mask_val = mask * offset

                mask_data = np.stack((*mask_coord, mask_val), 0).copy()
                flat_points = mask_data.reshape(3,-1).T

                clustering = DBSCAN(eps = 2, min_samples = 9).fit(flat_points)
                labels = clustering.labels_

                all_mask = []

                for i in np.unique(labels)[1:]:
                    if flat_points[np.where(labels == i)[0],-1].mean() / offset␣
↪< 245:
                        all_mask.append(np.where(labels == i, flat_points[:,-1]␣
↪/ offset, 0).reshape(*mask.shape[:2]))
                all_mask = np.array(all_mask)

                all_class = np.array(list(map(estimate_class, all_mask)))

                valid_class = all_class[np.where(all_class[:,0] < 245)[0]]
                valid_mask = all_mask[np.where(all_class[:,0] < 245)[0]]

                valid_mask = valid_mask[valid_class[:,1] < 3]
                valid_class = valid_class[valid_class[:,1] < 3]
                px_thresh = 10
                valid_class = valid_class[np.sum((valid_mask > 0.5).
↪astype(int), axis = (1,2)) > px_thresh]
                valid_mask = valid_mask[np.sum((valid_mask > 0.5).astype(int),␣
↪axis = (1,2)) > px_thresh]

                valid_mask = (valid_mask > 0.5).astype(int) * valid_class[:,0][:
↪,None, None].astype(int)

                bboxes = []
                # find bounding box
                for m in valid_mask:
                    # dim 0 is column position (x), dim 1 is row position (y)
                    box_coord = normalized_coords.transpose(1,2,0)[m > 0.5]
                    raw_box = np.quantile(box_coord, q = q, axis = 0)
                    b = [raw_box[0,0], raw_box[0,1], raw_box[1,0], raw_box[1,1]]
                    bboxes.append(b.copy())
```

```
                bboxes = np.array(bboxes)

                np.save(mask_file_name, (valid_mask > 0.5).astype(int))
                np.save(bbox_file_name, bboxes)
```

Data Loaders

We use a filename based data loader and keep the data for each image separately to avoid the memory overhead as in the homework when masks and labels are loaded all at once to the memory.

```
import torch
import torch.nn as nn
import torchvision
import PIL
import math
import os
import random
import numpy as np
from matplotlib import pyplot as plt


def train_test_split(DATA_DIR, seed = 0, train_ratio = 0.8):
    random.seed(seed)
    all_files = os.listdir(DATA_DIR + 'suboriginals')
    for i, f in enumerate(all_files):
        if len(f) <=  7:

            _ = all_files.pop(i)
            print('Irrelevant file found')
            break
    print('Clean completed')

    for i in range(len(all_files)):
        all_files[i] = all_files[i][:-4]

    split_idx = int(train_ratio * len(all_files))

    return all_files[:split_idx], all_files[split_idx:]
class data_loader:
    def __init__(self, names, DATA_DIR, batch_size, transforms):

        self.batch_size = batch_size
        self.idx = 0

        self.DATA_DIR = DATA_DIR

        image_folder = DATA_DIR + 'suboriginals/'
        mask_folder = DATA_DIR + 'submasks/'
        bboxes_folder = DATA_DIR + 'subbboxes/'
        self.transforms = transforms


        self.names = names

        random.shuffle(self.names)

        if len(self.names) % batch_size == 0:
            self.length = len(self.names) // batch_size
        else:
            self.length = len(self.names) // batch_size + 1

        self.mask_folder = mask_folder
        self.bboxes_folder = bboxes_folder
        self.image_folder = image_folder

    def __iter__(self):
        random.shuffle(self.names)
        self.idx = 0
        return self
    def __next__(self):
```

```
        if self.idx < self.length:

            batch = self.names[(self.idx * self.batch_size):((self.idx+1) * self.batch_size)]

            sequences = []
            bboxes = []
            masks = []
            for b in batch:
                i,m,n,t = [int(n) for n in b.split('-')]


                sequences.append(self.transforms(PIL.Image.open(self.image_folder + b + '.tif')))


                bbox = np.load(self.bboxes_folder + b + '.npy')
                mask = np.load(self.mask_folder + b + '.npy')
                bboxes.append(bbox)
                masks.append(mask)
            sequences = torch.stack(sequences)

            self.idx += 1

            return sequences, masks, bboxes, batch

        else:
            raise StopIteration
    def __len__(self):
        return self.length

class recurrent_loader:
    def __init__(self, names, DATA_DIR, batch_size, lags, transforms, time_padding = torch.zeros(1,313,313)):

        self.batch_size = batch_size
        self.idx = 0

        self.DATA_DIR = DATA_DIR

        image_folder = DATA_DIR + 'suboriginals/'
        mask_folder = DATA_DIR + 'submasks/'
        bboxes_folder = DATA_DIR + 'subbboxes/'
        self.time_padding = time_padding
        self.lags = lags
        self.transforms = transforms


        self.names = names

        random.shuffle(self.names)

        if len(self.names) % batch_size == 0:
            self.length = len(self.names) // batch_size
        else:
            self.length = len(self.names) // batch_size + 1

        self.mask_folder = mask_folder
        self.bboxes_folder = bboxes_folder
        self.image_folder = image_folder

    def __iter__(self):
        random.shuffle(self.names)
        self.idx = 0
        return self
```

```
    def __next__(self):
        if self.idx < self.length:

            batch = self.names[(self.idx * self.batch_size):((self.idx+1) * self.batch_size)]

            sequences = []
            bboxes = []
            masks = []
            for b in batch:
                i,m,n,t = [int(n) for n in b.split('-')]

                forward_images = []
                backward_images = []

                if i == 0:
                    max_t = 49
                else:
                    max_t = 18

                for time_idx in range(t - self.lags, t + 1):
                    if time_idx < 0 or time_idx > max_t:
                        forward_images.append(self.time_padding)
                    else:
                        forward_images.append(self.transforms(PIL.Image.open(self.image_folder + f'{i}-{m}-{n}-{time_idx}' + '.tif')))
                forward_images = torch.stack(forward_images)
                for time_idx in range(t, t + self.lags + 1):
                    if time_idx < 0 or time_idx > max_t:
                        backward_images.append(self.time_padding)
                    else:
                        backward_images.append(self.transforms(PIL.Image.open(self.image_folder + f'{i}-{m}-{n}-{time_idx}' + '.tif')))
                backward_images = torch.stack(backward_images)
                sequence = torch.stack((forward_images, backward_images))

                sequences.append(sequence.clone())


                bbox = np.load(self.bboxes_folder + b + '.npy')
                mask = np.load(self.mask_folder + b + '.npy')
                bboxes.append(bbox)
                masks.append(mask)
            sequences = torch.stack(sequences)

            self.idx += 1

            return sequences, masks, bboxes, batch

        else:
            raise StopIteration
    def __len__(self):
        return self.length


if __name__ == "__main__":

    transforms = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
                                                 torchvision.transforms.Grayscale(),
                                                 torchvision.transforms.Normalize([0.5], [0.5])])
    DATA_DIR = '../processdata/'
    train_names, test_names = train_test_split(DATA_DIR)
```

```
    train_loader = recurrent_loader(train_names, DATA_DIR, batch_size = 4, lags = 2, transforms = transforms)
    test_loader = recurrent_loader(test_names, DATA_DIR, batch_size = 4, lags = 2, transforms = transforms)


    for sequences, masks, bboxes, batch in iter(train_loader):
        forward_images, backward_images = sequences[0]
        plt.figure(figsize = (20,3))
        i = 0
        for im in forward_images[:-1]:
            i+=1
            plt.subplot(1,5,i)

            plt.imshow(im.permute(1,2,0), cmap = 'gray')
            plt.axis('off')
        for im in backward_images:
            i+=1
            plt.subplot(1,5,i)

            plt.imshow(im.permute(1,2,0), cmap = 'gray')
            plt.axis('off')
        plt.show()
        plt.close()

        plt.figure(figsize = (12,3))
        plt.subplot(1,3,1)
        plt.imshow(forward_images[-1][0],cmap = 'gray')
        plt.axis('off')
        plt.subplot(1,3,2)
        plt.imshow(np.array(PIL.Image.open(train_loader.DATA_DIR + 'sublabeled/' + batch[0] + '.tif')),cmap = 'gray')
        plt.axis('off')
        plt.subplot(1,3,3)
        plt.imshow(1 - masks[0].sum(0),cmap = 'gray')
        plt.axis('off')
        plt.show()
        plt.close()

        break
```

Architectures

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable

import os
import torch
from torch import nn
from torch.autograd import Variable

class GRU_cell(nn.Module):
    def __init__(self, in_channels, hidden_channels, gate_kernel):
        super(GRU_cell,self).__init__()

        self.in_channels = in_channels
        self.hidden_channels = hidden_channels

        self.Wz = nn.Conv2d(in_channels, hidden_channels, padding='same', bias = True, kernel_size = gate_kernel)
```

```
        self.Uz = nn.Conv2d(hidden_channels, hidden_channels, padding='same', bias = False, kernel_size = gate_kernel)

        self.Wr = nn.Conv2d(in_channels, hidden_channels, padding='same', bias = True, kernel_size = gate_kernel)
        self.Ur = nn.Conv2d(hidden_channels, hidden_channels, padding='same', bias = False, kernel_size = gate_kernel)

        self.Wh = nn.Conv2d(in_channels, hidden_channels, padding='same', bias = True, kernel_size = gate_kernel)
        self.Uh = nn.Conv2d(hidden_channels, hidden_channels, padding='same', bias = False, kernel_size = gate_kernel)

        self.sigmoid = nn.Sigmoid()
        # leaky relu to avoid vanishing gradient
        self.leaky_relu = nn.LeakyReLU()


    def forward(self,images, hidden_state):
        # N * C * W * H


        zt = self.sigmoid(self.Wz(images) + self.Uz(hidden_state))
        rt = self.sigmoid(self.Wr(images) + self.Ur(hidden_state))

        h_hat = self.leaky_relu(self.Wh(images) + self.Uh(rt * hidden_state))

        return (1 - zt) * h_hat + zt * hidden_state

class GRU(nn.Module):
    def __init__(self, in_channels, hidden_channels, gate_kernel):
        super(GRU,self).__init__()

        self.cell = GRU_cell(in_channels, hidden_channels, gate_kernel)
        self.hidden_channels = hidden_channels

    def forward(self, sequence, hidden_state = None):

        # N * T * C * W * H

        if hidden_state == None:
            hidden_state = torch.zeros(sequence.shape[0], self.hidden_channels, sequence.shape[-2], sequence.shape[-1]).to(sequence.device)

        for i in range(sequence.shape[1]):
            hidden_state = self.cell(sequence[:,i],hidden_state)

        return hidden_state

class bi_GRU(nn.Module):
    def __init__(self, in_channels, hidden_channels, gate_kernel = 3):
        super(bi_GRU,self).__init__()

        self.gru = GRU(in_channels, hidden_channels, gate_kernel)
        self.hidden_channels = hidden_channels
        self.bn = nn.BatchNorm2d(hidden_channels)

        # 1 * 1 convolution to combine information
        self.l = nn.Conv2d(2 * self.hidden_channels, hidden_channels, padding='same', bias = False, kernel_size = 1)

    def forward(self, sequences):

        # N * 2 * T * C * W * H
        lags = sequences.shape[2]
```

```
77
78
79          forward = self.gru(sequences[:,0])
80          bacward = self.gru(torch.flip(sequences[:,1], dims = [1]))
81
82
83          hidden_state = torch.cat([forward, bacward],1)
84
85          return self.bn(self.l(hidden_state))
86
87
88  class Bottleneck(nn.Module):
89      expansion = 4
90      def __init__(self, inplanes, planes, stride=1, downsample=None):
91          super(Bottleneck, self).__init__()
92          self.conv1 = nn.Conv2d(inplanes, planes, kernel_size=1, stride=stride,
        bias=False)
93          self.bn1 = nn.BatchNorm2d(planes)
94          self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1, padding=1,
         bias=False)
95          self.bn2 = nn.BatchNorm2d(planes)
96          self.conv3 = nn.Conv2d(planes, planes * 4, kernel_size=1, bias=False)
97          self.bn3 = nn.BatchNorm2d(planes * 4)
98          self.relu = nn.ReLU(inplace=True)
99          self.downsample = downsample
100         self.stride = stride
101     def forward(self, x):
102         residual = x
103         out = self.conv1(x)
104         out = self.bn1(out)
105         out = self.relu(out)
106         out = self.conv2(out)
107         out = self.bn2(out)
108         out = self.relu(out)
109         out = self.conv3(out)
110         out = self.bn3(out)
111         if self.downsample is not None:
112             residual = self.downsample(x)
113         out += residual
114         out = self.relu(out)
115         return out
116 class FPN(nn.Module):
117     def __init__(self, block, layers, input_channel = 1):
118         super(FPN, self).__init__()
119         self.in_planes = 64
120         self.conv1 = nn.Conv2d(input_channel, 64, kernel_size=7, stride=2, padding
        =3, bias=False)
121         self.bn1 = nn.BatchNorm2d(64)
122         # Bottom-up layers
123         self.layer1 = self._make_layer(block,  64, layers[0])
124         self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
125         self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
126         self.layer4 = self._make_layer(block, 512, layers[3], stride=2)
127         # Top layer
128         self.toplayer = nn.Conv2d(2048, 256, kernel_size=1, stride=1, padding=0)
        # Reduce channels
129         # Smooth layers
130         self.smooth1 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)
131         self.smooth2 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)
132         self.smooth3 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)
133         # Lateral layers
134         self.latlayer1 = nn.Conv2d(1024, 256, kernel_size=1, stride=1, padding=0)
135         self.latlayer2 = nn.Conv2d( 512, 256, kernel_size=1, stride=1, padding=0)
136         self.latlayer3 = nn.Conv2d( 256, 256, kernel_size=1, stride=1, padding=0)
137
```

```
    def _make_layer(self, block, planes, blocks, stride=1):
        downsample = None
        if stride != 1 or self.in_planes != planes * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.in_planes, planes * block.expansion,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(planes * block.expansion),
            )
        layers = []
        layers.append(block(self.in_planes, planes, stride, downsample))
        self.in_planes = planes * block.expansion
        for i in range(1, blocks):
            layers.append(block(self.in_planes, planes))
        return nn.Sequential(*layers)
    def _upsample_add(self, x, y):
        _,_,H,W = y.size()
        return F.interpolate(x, size=(H,W), mode='bilinear',align_corners=True) +
    y
    def forward(self, x):
        # Bottom-up
        c1 = F.relu(self.bn1(self.conv1(x)))
        c1 = F.max_pool2d(c1, kernel_size=3, stride=2, padding=1)
        #print(f'c1:{c1.shape}')
        c2 = self.layer1(c1)
        #print(f'c2:{c2.shape}')
        c3 = self.layer2(c2)
        #print(f'c3:{c3.shape}')
        c4 = self.layer3(c3)
        #print(f'c4:{c4.shape}')
        c5 = self.layer4(c4)
        #print(f'c5:{c5.shape}')
        # Top-down
        p5 = self.toplayer(c5)
        #print(f'p5:{p5.shape}')
        p4 = self._upsample_add(p5, self.latlayer1(c4))
        #print(f'latlayer1(c4):{self.latlayer1(c4).shape}, p4:{p4.shape}')
        p3 = self._upsample_add(p4, self.latlayer2(c3))
        #print(f'latlayer1(c3):{self.latlayer2(c3).shape}, p3:{p3.shape}')
        p2 = self._upsample_add(p3, self.latlayer3(c2))
        #print(f'latlayer1(c2):{self.latlayer3(c2).shape}, p2:{p2.shape}')
        # Smooth
        p4 = self.smooth1(p4)
        p3 = self.smooth2(p3)
        p2 = self.smooth3(p2)
        return p2, p3, p4, p5
    def sequence_output(self,sequence):
        # sequence: (batchsize, direction, time, channels, height, width)

        direction = sequence.shape[1]
        batch = sequence.shape[0]
        out = [[],[],[],[]]

        for i in range(batch):
            forward = self(sequence[i,0])
            backward = self(sequence[i,1])

            for j,(f,b) in enumerate(zip(forward, backward)):
                out[j].append(torch.stack([f, b]))


        return [torch.stack(o) for o in out]



class SOLO_head(nn.Module):
```

```
    def __init__(self,
                 scale_ranges,
                 num_grids,
                 mask_loss_cfg,
                 cate_loss_cfg,
                 postprocess_cfg):
        super(SOLO_head, self).__init__()
        self.scale_ranges = scale_ranges
        print(scale_ranges)
        self.num_grids = num_grids
        print(num_grids)
        self.mask_loss_cfg = mask_loss_cfg
        self.cate_loss_cfg = cate_loss_cfg
        self.postprocess_cfg = postprocess_cfg

        self.bce_loss = nn.BCELoss()

        self.cate_branch = []
        for i in range(7):
            self.cate_branch.append(nn.Conv2d(256, 256, 3, 1, 1, bias=False))
            self.cate_branch.append(nn.GroupNorm(32, 256))
            self.cate_branch.append(nn.ReLU())
        self.cate_branch.append(nn.Conv2d(256, 1, 3, 1, 1, bias=True))
        self.cate_branch.append(nn.Sigmoid())
        self.cate_branch = nn.Sequential(*self.cate_branch)

        self.ins_branch = []
        self.ins_branch.append(nn.Conv2d(258, 256, 3, 1, 1, bias=False))
        self.ins_branch.append(nn.GroupNorm(32, 256))
        self.ins_branch.append(nn.ReLU())
        for j in range(6):
            self.ins_branch.append(nn.Conv2d(256, 256, 3, 1, 1, bias=False))
            self.ins_branch.append(nn.GroupNorm(32, 256))
            self.ins_branch.append(nn.ReLU())
        self.ins_branch = nn.Sequential(*self.ins_branch)

        self.ins_outs = nn.ModuleList()
        for i in range(len(self.num_grids)):
            self.ins_outs.append(nn.Sequential(nn.Conv2d(256, self.num_grids[i
]**2, 1, padding=0, bias=True), nn.Sigmoid()))



    def generate_single_target(self, single_bboxes, single_labels, single_mask,
featmap_sizes, eps, device='cpu'):

        active_masks = []
        cate_masks = []
        target_masks = []

        center_x1 = ((1 + eps) / 2 * single_bboxes[:, 0] + (1 - eps) / 2 *
single_bboxes[:, 2])
        center_x2 = ((1 - eps) / 2 * single_bboxes[:, 0] + (1 + eps) / 2 *
single_bboxes[:, 2])

        center_y1 = ((1 + eps) / 2 * single_bboxes[:, 1] + (1 - eps) / 2 *
single_bboxes[:, 3])
        center_y2 = ((1 - eps) / 2 * single_bboxes[:, 1] + (1 + eps) / 2 *
single_bboxes[:, 3])
        target_scales = torch.sqrt((single_bboxes[:, 2] - single_bboxes[:, 0]) * (
single_bboxes[:, 3] - single_bboxes[:, 1]))

        for fp_size, num_grid, fpn_scale in zip(featmap_sizes, self.num_grids,
self.scale_ranges):
            new_x1 = center_x1 * num_grid
```

```
            new_x2 = center_x2 * num_grid
            new_y1 = center_y1 * num_grid
            new_y2 = center_y2 * num_grid

            cate_mask = torch.zeros(num_grid, num_grid)
            target_mask = torch.zeros(num_grid ** 2, fp_size[0]*2,fp_size[0]*2)

            for obj_idx in range(len(single_labels)):

                # Check scale range
                if fpn_scale[0] <= target_scales[obj_idx].item() < fpn_scale[1]:

                    top, bottom, left, right = new_y1[obj_idx].int().item(), new_y2[obj_idx].int().item(), new_x1[obj_idx].int().item(), new_x2[obj_idx].int().item()

                    if top >= bottom:
                        top = int(((top + bottom) / 2))
                        bottom = top + 1
                    if left >= right:
                        left = int(((left + right) / 2))
                        right = left + 1
                    cate_mask[top:bottom, left:right] = single_labels[obj_idx]
                    temp_mask = single_mask[obj_idx][None, None, :]

                    interp_mask = nn.functional.interpolate(temp_mask, size=[2 * f for f in fp_size], mode='bilinear', align_corners = True)[0,0]
                    interp_mask = (interp_mask > 0.5).int().clone()

                    pos = torch.arange(top,bottom).reshape(-1, 1) * num_grid + torch.arange(left, right)
                    pos = pos.flatten().int()
                    for p in pos:
                        target_mask[p] = interp_mask

            active_mask = (cate_mask > 0)
            active_masks.append(active_mask.bool().flatten().to(device))
            cate_masks.append(cate_mask.int().flatten().to(device))
            target_masks.append(target_mask.to(device))

        return target_masks, cate_masks, active_masks

    def generate_targets(self, gt_masks, gt_bboxes, gt_labels, eps, featmap_sizes=None, device='cpu'):
        if featmap_sizes is None:
            featmap_sizes = self.featmap_sizes
        cate_targets = []
        mask_targets = []
        active_masks = []

        for bbox, lab, mask in zip(gt_bboxes, gt_labels, gt_masks):
            per_img_result = self.generate_single_target(bbox, lab, mask, featmap_sizes=featmap_sizes, eps=eps, device=device)
            mask_targets.append(per_img_result[0])
            cate_targets.append(per_img_result[1])
            active_masks.append(per_img_result[2])
        return mask_targets, cate_targets, active_masks

    def loss(self,
             category_predictions,
             mask_predictions,
             mask_targets,
             cate_targets,
             active_masks,
             device='cpu'):
```

```
318
319         mask_trues, mask_preds, cate_trues, cate_preds = [], [], [], []
320
321         for target_level, active_level in zip(zip(*mask_targets), zip(*
    active_masks)):
322             tmp = []
323             for target_image, active_image in zip(target_level, active_level):
324                 tmp.append(target_image[active_image, :, :])
325             mask_trues.append(torch.cat(tmp, axis=0))
326
327         for pred_level, active_level in zip(mask_predictions, zip(*active_masks)):
328             tmp = []
329             for pred_image, active_image in zip(pred_level, active_level):
330                 tmp.append(pred_image[active_image, :, :])
331             mask_preds.append(torch.cat(tmp, axis=0))
332
333         for level in zip(*cate_targets):
334             for image in level:
335                 cate_trues += image.flatten().tolist()
336         cate_trues = torch.Tensor(cate_trues).to(device)
337
338         for level in category_predictions:
339             cate_preds.append(level.flatten())
340         cate_preds = torch.cat(cate_preds).to(device)
341
342
343         cate_loss = self.bce_loss(cate_preds,cate_trues)
344 #         cate_loss = self.get_cate_loss(cate_trues, cate_preds, device=device)
345         mask_loss = self.get_mask_loss(mask_trues, mask_preds, device=device)
346
347
348         lambda_cate, lambda_mask = self.cate_loss_cfg['weight'], self.
    mask_loss_cfg['weight']
349         total_loss = lambda_cate * cate_loss + lambda_mask * mask_loss
350
351         return cate_loss, mask_loss, total_loss
352
353     def dice_loss(self, y_true, y_pred, smooth=1e-6, device='cpu'):
354         y_true, y_pred = y_true.to(device), y_pred.to(device)
355         intersection = 2 * torch.sum(y_true * y_pred)
356         sum_true = torch.sum(y_true * y_true)
357         sum_pred = torch.sum(y_pred * y_pred)
358         dice_coeff = (intersection + smooth) / (sum_true + sum_pred + smooth)
359         return 1. - dice_coeff
360
361     def get_mask_loss(self, mask_trues, mask_preds, device='cpu'):
362         '''Loss function of mask'''
363         num_pos = sum([level.shape[0] for level in mask_preds])  # the number of
    positive samples
364         summation = sum([sum([self.dice_loss(true, pred)
365                               for true, pred in zip(true_level, pred_level)])
366                               for true_level, pred_level in zip(mask_trues,
    mask_preds)
367                               if pred_level.shape[0] > 0])
368         return summation / num_pos if num_pos != 0 else torch.tensor(0)
369
370     def points_nms(self, heat, kernel=2):
371         '''
372         Credit to solo's auther: https://github.com/WXinlong/SOLO
373         Input:
374             - heat: (batch_size, C-1, S, S)
375         Output:
376                     (batch_size, C-1, S, S)
377         '''
378         # 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

    def matrix_nms(self, sorted_masks, sorted_scores, method='gauss', gauss_sigma
    =0.5):
        ''' Performs Matrix NMS
        Input:
            - sorted_masks: (n_active, image_h/4, image_w/4)
            - sorted_scores: (n_active,)
        Output:
            - decay_scores: (n_active,)
        '''
        n = len(sorted_scores)
        sorted_masks = sorted_masks.reshape(n, -1)
        intersection = torch.mm(sorted_masks, sorted_masks.T)
        areas = sorted_masks.sum(dim=1).expand(n, n)
        union = areas + areas.T - intersection
        ious = (intersection / union).triu(diagonal=1)

        ious_cmax = ious.max(0)[0].expand(n, n).T
        if method == 'gauss':
            decay = torch.exp(-(ious ** 2 - ious_cmax ** 2) / gauss_sigma)
        else:
            decay = (1 - ious) / (1 - ious_cmax)
        decay = decay.min(dim=0)[0]
        return sorted_scores * decay

    def post_processing(self, cate_pred, mask_pred, nms_score_threshold = 0.1):

        cate_thresh = self.postprocess_cfg['cate_thresh']
        ins_thresh = self.postprocess_cfg['ins_thresh']
        pre_NMS_num = self.postprocess_cfg["pre_NMS_num"]
        keep_instance = self.postprocess_cfg['keep_instance']

        cate_flat = torch.cat([single[0].flatten() for single in cate_pred]).cpu()
        mask_flat = torch.cat([single[0] for single in mask_pred]).cpu()

        cate_score_all = cate_flat
        cate_all = torch.ones_like(cate_flat)
        mask_flat_all = mask_flat

        cate_score = cate_score_all[cate_score_all > cate_thresh][:pre_NMS_num]
        cate_label = cate_all[cate_score_all > cate_thresh][:pre_NMS_num]
        masks = mask_flat_all[cate_score_all > cate_thresh][:pre_NMS_num]

        binary_mask = (masks > ins_thresh).int()

        mask_score = torch.sum(torch.where(masks > ins_thresh, masks, torch.tensor
    (0.)), (1,2)) / torch.sum(binary_mask, (1,2))
        final_score = mask_score * cate_score

        sort_idx = torch.argsort(final_score, descending=True)
        sorted_binary_mask = binary_mask[sort_idx]
        sort_final_score = final_score[sort_idx]
        sorted_cate = cate_label[sort_idx]
        if len(sorted_binary_mask) < 1:
            return torch.zeros(0, *mask_flat.shape[-2:]), torch.tensor([]), torch.
    tensor([])

        nms_score = self.matrix_nms(sorted_binary_mask, sort_final_score)

        nms_sort_idx = torch.argsort(nms_score, descending=True)[:keep_instance]
        nms_sorted_final_score = nms_score[nms_sort_idx]
```

```
        nms_binary_mask = sorted_binary_mask[nms_sort_idx][nms_sorted_final_score > nms_score_threshold]
        nms_label = sorted_cate[nms_sort_idx][nms_sorted_final_score > nms_score_threshold]


        return nms_binary_mask, nms_label, sort_final_score[nms_sort_idx]
```

Optional Configuration file for training

```
eps = 0.2

# num_grids = [60, 40, 20, 20]
# scale_ranges = ((0, 0.4), (0.2, 0.6), (0.4, 0.8), (0.6, 1))

# scale_ranges = ((0, 0.3), (0.1, 0.4), (0.3, 0.7), (0.6, 1))
# num_grids = [80, 60, 30, 10]

# scale_ranges = ((0.05, 0.4), (0.1, 0.5), (0.4, 0.7), (0.6, 1))
# num_grids = [90, 60, 30, 10]

num_grids = [80, 40, 20, 10]
scale_ranges = ((0, 0.25), (0.2, 0.5), (0.4, 0.8), (0.6, 1))

batch_size = 4

mask_weight = 1.
cate_weight = 1.
```

SOLO Training

```

# %%
from mpl_toolkits.mplot3d import Axes3D
from sklearn.cluster import DBSCAN
from tqdm import tqdm, trange
import random
import numpy as np
import torch
import torchvision
from torchvision import transforms
import torch.nn.functional as F
from torch import nn
from torch.utils.data import Dataset, DataLoader
from torchvision.models.detection import maskrcnn_resnet50_fpn
import h5py
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import matplotlib.patches as patches
from PIL import Image
import os
from IPython.display import clear_output
import math
import matplotlib.pyplot as plt
import PIL
import timeit
import os
import ast

import warnings
warnings.filterwarnings("ignore")

store_folder = input('checkpoints folder name: ')

```

```
checkpoint_path = f'../check_points/{store_folder}'

if not os.path.isdir(checkpoint_path):
    os.mkdir(checkpoint_path)

print('Checkpoints path : ', checkpoint_path)
# %%
draw_box = lambda x1, y1, x2, y2: np.array([[x1, y1], [x1, y2], [x2, y2], [x2, y1], [x1, y1]])

# %% [markdown]
# # Load Data
if_load = input('Load presets? T/F')[0]
if  if_load in ['t', 'T']:
    from config import *
else:
# %%
    num_grids = input('num_grids:')
    num_grids = ast.literal_eval(num_grids)
    num_grids = [int(f) for f in num_grids]


    scale_ranges = input('scale_ranges: ')
    scale_ranges = ast.literal_eval(scale_ranges)
    scale_ranges = [[float(e) for e in f] for f in scale_ranges]
    batch_size = int(input('batch_size: '))
    mask_weight = float(input("mask_weight: "))
    cate_weight = float(input("cate_weight: "))
    eps = float(input("eps: "))


# ((0, 0.4), (0.1, 0.5), (0.3, 0.7), (0.6, 1))
from data_loader import *
DATA_DIR = "../processdata/"
test_files = list(np.load("test_filenames.npy"))
train_files = list(np.load("train_filenames.npy"))

transforms = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
                                             torchvision.transforms.Grayscale(),
                                             torchvision.transforms.Normalize([0.5], [0.5])])
train_loader = data_loader(train_files, DATA_DIR, batch_size = batch_size, transforms = transforms)
test_loader = data_loader(test_files, DATA_DIR, batch_size = batch_size,transforms = transforms)

# %% [markdown]
# ## Visual Inspection

# %%
# data = next(iter(train_loader))
for data in train_loader:
    break
images, masks, bboxes, batch = data


# %% [markdown]
# # Model

# %%
# loading existing configuration
from architecture import *
# from config import *

```

```
postprocess_cfg = dict(cate_thresh=0.5,
                           ins_thresh=0.5,
                           pre_NMS_num=100,
                           # pre_NMS_num=50,
                           keep_instance=10)
                           # keep_instance=15)
nms_score_threshold = 0.5

# %%
class SOLO(SOLO_head):
    def __init__(self,
                 scale_ranges=scale_ranges,
                 num_grids=num_grids,
                 mask_loss_cfg=dict(weight=mask_weight),
                 cate_loss_cfg=dict(gamma=2,
                                    alpha=0.25,
                                    weight=cate_weight),
                 postprocess_cfg=postprocess_cfg):
        super(SOLO, self).__init__(scale_ranges, num_grids, mask_loss_cfg, cate_loss_cfg, postprocess_cfg)


        self.backbone = FPN(Bottleneck, [3,4,6,3])

    def forward(self, images, device, eval=False, original_img = None):

        feature_pyramid = list(self.backbone(images))

        category_predictions = []
        mask_predictions = []

        if eval == True:

            for i, v in enumerate(feature_pyramid):
                num_grid = self.num_grids[i]
                category_pred = nn.functional.interpolate(v, size=[num_grid, num_grid], mode="bilinear", align_corners = True)
                category_pred = self.cate_branch(category_pred)
                category_predictions.append(self.points_nms(category_pred).permute(0,2,3,1))

                meshxy = torch.stack(torch.meshgrid(torch.linspace(-1,1,v.shape[-2]), torch.linspace(-1,1,v.shape[-1])))
                batch_meshxy = torch.tile(meshxy, (images.shape[0],1,1,1)).to(device)
                new_v = torch.cat((v, batch_meshxy), 1)

                mask_pred = self.ins_outs[i](self.ins_branch(new_v))

                mask_predictions.append(nn.functional.interpolate(mask_pred, size=original_img, mode='bilinear', align_corners = True))
            return category_predictions, mask_predictions

        else:
            for i, v in enumerate(feature_pyramid):
                num_grid = self.num_grids[i]
                category_pred = nn.functional.interpolate(v, size=[num_grid, num_grid], mode="bilinear", align_corners = True)
                category_predictions.append(self.cate_branch(category_pred))

                meshxy = torch.stack(torch.meshgrid(torch.linspace(-1,1,v.shape[-2]), torch.linspace(-1,1,v.shape[-1])))
                batch_meshxy = torch.tile(meshxy, (images.shape[0],1,1,1)).to(device)
                new_v = torch.cat((v, batch_meshxy), 1)
```

```
                mask_pred = self.ins_outs[i](self.ins_branch(new_v))

                mask_predictions.append(nn.functional.interpolate(mask_pred, scale_factor=2, mode='bilinear', align_corners = True))
            return category_predictions, mask_predictions


# %%



print({'num_grids': num_grids, 'scale_ranges': scale_ranges, "eps":eps})

# %%
device = 'cuda:0'
model = SOLO().to(device)
backbone_out = model.backbone(torch.zeros(0,1,313,313).to(device))
model.featmap_sizes = [list(f.shape[-2:]) for f in backbone_out]


# %%

optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
# optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)
np.save(f'{checkpoint_path}/hyper_parameters.npy', {'num_grids': num_grids, 'scale_ranges': scale_ranges, "eps":eps, "optimizer" : type(optimizer).__name__, 'mask_weight':mask_weight, 'cate_weight':cate_weight}, allow_pickle = True)
scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=[15, 30, 45, 60, 75], gamma=0.95)

num_epoch = 75
print(f"batch size : {batch_size}, epoch : {num_epoch}")
loss_train_stor = []
loss_val_stor = []

model.postprocess_cfg = postprocess_cfg


# %%
# np.save('train_filenames.npy', train_files, allow_pickle = True)
# np.save('test_filenames.npy', test_files, allow_pickle = True)
for epoch in tqdm(range(num_epoch)):



    model.train()
    print({'num_grids': num_grids, 'scale_ranges': scale_ranges, "eps":eps, 'cate_weight':cate_weight, 'mask_weight' : mask_weight})

    print(f'\n== EPOCH {epoch+1}; Loss:  Cate      Mask      Total  Elapsed time')

    # Train
    start_time = timeit.default_timer()
    for iter, data in enumerate(train_loader):

        optimizer.zero_grad()

        # load data and adjust dtype
        images, masks, bboxes, batch = data
        masks, bboxes = [torch.Tensor(m) for m in masks], [torch.Tensor(b) for b in bboxes]
        images = images.to(device)
        labels = [torch.ones(b.shape[0]).to(device) for b in bboxes]

```

```
        targets = model.generate_targets(masks, bboxes, labels, device = device, eps = eps)
        out = model(images.to(device), device = device)
        cate_loss_train, mask_loss_train, loss_train = model.loss(*out, *targets, device)
        loss_train_stor.append((cate_loss_train.cpu().detach().item(),
                                mask_loss_train.cpu().detach().item(),
                                loss_train.cpu().detach().item()))
        loss_train.backward()
        optimizer.step()
        if iter % 10 == 0:
            print(f"Train Iter: {iter:>3}; {cate_loss_train.cpu().detach().item():.2e}, {mask_loss_train.cpu().detach().item():.2e}, {loss_train.cpu().detach().item():.2e}, {timeit.default_timer() - start_time:.1f}")

    scheduler.step()
    start_time = timeit.default_timer()

    with torch.no_grad():
        model.eval()

        for iter, data in enumerate(test_loader):

            optimizer.zero_grad()

            # load data and adjust dtype
            images, masks, bboxes, batch = data
            masks, bboxes = [torch.Tensor(m) for m in masks], [torch.Tensor(b) for b in bboxes]
            images = images.to(device)
            labels = [torch.ones(b.shape[0]).to(device) for b in bboxes]

            targets = model.generate_targets(masks, bboxes, labels, eps = eps, device = device)
            out = model(images.to(device), device = device)
            cate_loss_val, mask_loss_val, loss_val = model.loss(*out, *targets, device)
            loss_val_stor.append((cate_loss_val.cpu().detach().item(),
                                  mask_loss_val.cpu().detach().item(),
                                  loss_val.cpu().detach().item()))

            if iter % 10 == 0:
                print(f"Test Iter: {iter:>3}; {cate_loss_val.cpu().detach().item():.2e}, {mask_loss_val.cpu().detach().item():.2e}, {loss_val.cpu().detach().item():.2e}, {timeit.default_timer() - start_time:.1f}")

    path = f'{checkpoint_path}/solo-epoch-{epoch+1}.pth'
    torch.save({'epoch': epoch+1,
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict()
                }, path)


    np.save(f'{checkpoint_path}/loss-train-epoch-{epoch+1}.npy', np.array(loss_train_stor))
    np.save(f'{checkpoint_path}/loss-val-epoch-{epoch+1}.npy', np.array(loss_val_stor))

    if epoch+1 < num_epoch: # not the last epoch
        clear_output()
```

SOLO with GRU Training

```
\
```

```
# %%
from mpl_toolkits.mplot3d import Axes3D
from sklearn.cluster import DBSCAN
from tqdm import tqdm, trange
import random
import numpy as np
import torch
import torchvision
from torchvision import transforms
from torch import nn
import os
from IPython.display import clear_output
import timeit
import ast

import warnings
warnings.filterwarnings("ignore")

store_folder = input('checkpoints folder name: ')

checkpoint_path = f'../check_points/{store_folder}'

if not os.path.isdir(checkpoint_path):
    os.mkdir(checkpoint_path)

print('Checkpoints path : ', checkpoint_path)
# %%
draw_box = lambda x1, y1, x2, y2: np.array([[x1, y1], [x1, y2], [x2, y2], [x2, y1], [x1, y1]])

# %% [markdown]
# # Load Data
if_load = input('Load presets? T/F')[0]
if  if_load in ['t', 'T']:
    from config import *
else:
# %%
    num_grids = input('num_grids:')
    num_grids = ast.literal_eval(num_grids)
    num_grids = [int(f) for f in num_grids]


    scale_ranges = input('scale_ranges: ')
    scale_ranges = ast.literal_eval(scale_ranges)
    scale_ranges = [[float(e) for e in f] for f in scale_ranges]
    batch_size = int(input('batch_size: '))
    mask_weight = float(input("mask_weight: "))
    cate_weight = float(input("cate_weight: "))
    eps = float(input("eps: "))
from data_loader import *
DATA_DIR = "../processdata/"
train_files, test_files = train_test_split(DATA_DIR)
test_files = list(np.load("test_filenames.npy"))
train_files = list(np.load("train_filenames.npy"))
batch_size = 4
transforms = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
                                             torchvision.transforms.Grayscale(),
                                             torchvision.transforms.Normalize([0.5], [0.5])])
train_loader = recurrent_loader(train_files, DATA_DIR, batch_size = batch_size, transforms = transforms, lags = 2)
test_loader = recurrent_loader(test_files, DATA_DIR, batch_size = batch_size, transforms = transforms, lags = 2)

```

```
# %% [markdown]
# ## Visual Inspection

# %%
# data = next(iter(train_loader))
for data in train_loader:
    break
sequences, masks, bboxes, batch = data



# %%
from architecture import *
postprocess_cfg = dict(cate_thresh=0.5,
                           ins_thresh=0.5,
                           pre_NMS_num=100,
                           # pre_NMS_num=50,
                           keep_instance=10)
                           # keep_instance=15)

# %%


# %% [markdown]
# ## SOLO

# %%
class SOLO(SOLO_head):
    def __init__(self,
                 scale_ranges=scale_ranges,
                 num_grids=num_grids,
                 mask_loss_cfg=dict(weight=mask_weight),
                 cate_loss_cfg=dict(gamma=2,
                                    alpha=0.25,
                                    weight=cate_weight),
                 postprocess_cfg=postprocess_cfg):
        super(SOLO, self).__init__(scale_ranges, num_grids, mask_loss_cfg,
    cate_loss_cfg, postprocess_cfg)


        self.backbone = FPN(Bottleneck, [3,4,6,3])
        self.grus = nn.ModuleList([])

        for i in range(4):
            self.grus.append(bi_GRU(256,256,3))

    def forward(self, sequences, device, eval=False, original_img = None):

        sequences_feature = self.backbone.sequence_output(sequences)
        feature_pyramid = []

        for i in range(4):
            feature_pyramid.append(self.grus[i](sequences_feature[i]))







        category_predictions = []
        mask_predictions = []

        if eval == True:

            for i, v in enumerate(feature_pyramid):
                num_grid = self.num_grids[i]
```

```
                category_pred = nn.functional.interpolate(v, size=[num_grid, num_grid], mode="bilinear", align_corners = True)
                category_pred = self.cate_branch(category_pred)
                category_predictions.append(self.points_nms(category_pred).permute(0,2,3,1))

                meshxy = torch.stack(torch.meshgrid(torch.linspace(-1,1,v.shape[-2]), torch.linspace(-1,1,v.shape[-1])))
                batch_meshxy = torch.tile(meshxy, (sequences.shape[0],1,1,1)).to(device)
                new_v = torch.cat((v, batch_meshxy), 1)

                mask_pred = self.ins_outs[i](self.ins_branch(new_v))

                mask_predictions.append(nn.functional.interpolate(mask_pred, size=original_img, mode='bilinear', align_corners = True))
            return category_predictions, mask_predictions

        else:
            for i, v in enumerate(feature_pyramid):
                num_grid = self.num_grids[i]
                category_pred = nn.functional.interpolate(v, size=[num_grid, num_grid], mode="bilinear", align_corners = True)
                category_predictions.append(self.cate_branch(category_pred))

                meshxy = torch.stack(torch.meshgrid(torch.linspace(-1,1,v.shape[-2]), torch.linspace(-1,1,v.shape[-1])))
                batch_meshxy = torch.tile(meshxy, (sequences.shape[0],1,1,1)).to(device)
                new_v = torch.cat((v, batch_meshxy), 1)

                mask_pred = self.ins_outs[i](self.ins_branch(new_v))

                mask_predictions.append(nn.functional.interpolate(mask_pred, scale_factor=2, mode='bilinear', align_corners = True))
            return category_predictions, mask_predictions

# %%
device = 'cuda:0'
model = SOLO().to(device)
backbone_out = model.backbone(torch.zeros(0,1,313,313).to(device))
model.featmap_sizes = [list(f.shape[-2:]) for f in backbone_out]
labels = [torch.ones(b.shape[0]) for b in bboxes]
masks, bboxes = [torch.Tensor(m) for m in masks], [torch.Tensor(b) for b in bboxes]
target = model.generate_single_target(bboxes[0], labels[0], masks[0], model.featmap_sizes, eps = eps)

# %%
targets = model.generate_targets(masks, bboxes, labels, eps = eps, device = device)
out = model(sequences.to(device), device = device)


optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
# optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)
np.save(f'{checkpoint_path}/hyper_parameters.npy', {'num_grids': num_grids, 'scale_ranges': scale_ranges, "eps":eps, "optimizer" : type(optimizer).__name__, 'mask_weight':mask_weight, 'cate_weight':cate_weight}, allow_pickle = True)
scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=[15, 30, 45, 60, 75], gamma=0.95)

num_epoch = 75
print(f"batch size : {batch_size}, epoch : {num_epoch}")
loss_train_stor = []
```

```
loss_val_stor = []


model.postprocess_cfg = postprocess_cfg


for epoch in tqdm(range(num_epoch)):

    batch = 0
    n_show = 3
    model.eval()
    for i, data in enumerate(train_loader):
    # for i, data in enumerate(test_loader):

        sequences, masks, bboxes, _ = data
        masks, bboxes = [torch.Tensor(m) for m in masks], [torch.Tensor(b) for b in bboxes]
        original_img= [int(313/2),int(313/2)]

        post_out = model.post_processing(*model.forward(sequences.to(device),
                                                        device=device,
                                                        eval=True,
                                                        original_img=original_img),
                                         nms_score_threshold = 5e-1)

        plt.figure(figsize = (18,6))

        plt.subplot(1, 3, 1)
        plt.title('Raw Image')
        plt.imshow(sequences[batch][0][-1].squeeze(), cmap = 'gray')
        plt.axis('off')

        plt.subplot(1, 3, 2)
        plt.title(f'Ground Truth: {len(bboxes[batch])} instances')
        plt.imshow(sequences[batch][0][-1].squeeze(), cmap = 'gray')
        for box, msk in zip(bboxes[batch], masks[batch]):
            plt.plot(*draw_box(*box * 313).T, color = 'aqua')
            plt.imshow(msk, alpha = msk * 0.5, cmap = 'Reds')
        plt.axis('off')

        plt.subplot(1, 3, 3)
        plt.title(f'Prediction: {len(post_out[batch])} instances')
        plt.imshow(sequences[batch][0][-1].squeeze(), cmap = 'gray')
        for msk in post_out[0]:
            m = (nn.functional.interpolate(msk[None, None].float(), size=sequences.shape[-2:], mode='bilinear') > 0.5)[0,0].int().cpu().numpy()
            plt.imshow(m, alpha = m * 0.5, cmap = 'winter')
        plt.axis('off')
        if i == n_show-1:
            plt.savefig(f'{checkpoint_path}/solo-epoch-{epoch}-train-result.pdf', dpi = 100)
        plt.show()
        plt.close()

        if i == n_show-1:
            break

    # plt.savefig(f'{checkpoint_path}/train_outputs_epoch_{epoch + 1}.pdf', dpi = 100)


    model.train()
    print(f'Eps : {eps:.1f}')
```

```
    print({'num_grids': num_grids, 'scale_ranges': scale_ranges, "eps":eps, '
cate_weight':cate_weight, 'mask_weight' : mask_weight})

    print(f'\n== EPOCH {epoch+1}; Loss:  Cate      Mask      Total  Elapsed time')

    # Train
    start_time = timeit.default_timer()
    for iter, data in enumerate(train_loader):

        optimizer.zero_grad()

        # load data and adjust dtype
        sequences, masks, bboxes, batch = data
        masks, bboxes = [torch.Tensor(m) for m in masks], [torch.Tensor(b) for b
in bboxes]
        sequences = sequences.to(device)
        labels = [torch.ones(b.shape[0]).to(device) for b in bboxes]

        targets = model.generate_targets(masks, bboxes, labels, eps = eps, device
= device)
        out = model(sequences.to(device), device = device)
        cate_loss_train, mask_loss_train, loss_train = model.loss(*out, *targets,
device)
        loss_train_stor.append((cate_loss_train.cpu().detach().item(),
                                mask_loss_train.cpu().detach().item(),
                                loss_train.cpu().detach().item()))
        loss_train.backward()
        optimizer.step()
        if iter % 10 == 0:
            print(f"Train Iter: {iter:>3}; {cate_loss_train.cpu().detach().item()
:.2e}, {mask_loss_train.cpu().detach().item():.2e}, {loss_train.cpu().detach().
item():.2e}, {timeit.default_timer() - start_time:.1f}")

    scheduler.step()
    start_time = timeit.default_timer()

    with torch.no_grad():
        model.eval()

        for iter, data in enumerate(test_loader):

            optimizer.zero_grad()

            # load data and adjust dtype
            sequences, masks, bboxes, batch = data
            masks, bboxes = [torch.Tensor(m) for m in masks], [torch.Tensor(b) for
 b in bboxes]
            sequences = sequences.to(device)
            labels = [torch.ones(b.shape[0]).to(device) for b in bboxes]

            targets = model.generate_targets(masks, bboxes, labels, eps = eps,
device = device)
            out = model(sequences.to(device), device = device)
            cate_loss_val, mask_loss_val, loss_val = model.loss(*out, *targets,
device)
            loss_val_stor.append((cate_loss_val.cpu().detach().item(),
                                  mask_loss_val.cpu().detach().item(),
                                  loss_val.cpu().detach().item()))

            if iter % 10 == 0:
                print(f"Test Iter: {iter:>3}; {cate_loss_val.cpu().detach().item()
:.2e}, {mask_loss_val.cpu().detach().item():.2e}, {loss_val.cpu().detach().item
():.2e}, {timeit.default_timer() - start_time:.1f}")

    path = f'{checkpoint_path}/solo-epoch-{epoch+1}.pth'
```

```
    torch.save({'epoch': epoch+1,
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict()
                }, path)


    np.save(f'{checkpoint_path}/loss-train-epoch-{epoch+1}.npy', np.array(loss_train_stor))
    np.save(f'{checkpoint_path}/loss-val-epoch-{epoch+1}.npy', np.array(loss_val_stor))

    if epoch+1 < num_epoch: # not the last epoch
        clear_output()
```

Average Precision

```
import torch
from collections import Counter


def mean_average_precision(pred_masks, true_masks, iou_threshold = 0.5, num_classes = 1):
    average_precision = 0
    epsilon = 1e-6

    detections = []
    ground_truths = []

    for detection in pred_masks:
        detections.append(detection)
    for true_mask in true_masks:
        ground_truths.append(true_mask)
    #img 0 has 3 masks
    #img 1 has 5 masks
    #amount_masks = {0:3, 1:5}
    amount_masks = Counter([gt[0] for gt in ground_truths])
    for key, val in amount_masks.items():
        #amount_boxes = {0:torch.tensor([0,0,0]), 1:torch.tensor([0,0,0,0,0])}
        amount_masks[key] = torch.zeros(val)
        detections.sort(key = lambda x: x[1], reverse = True)
        TP = torch.zeros((len(detections)))
        FP = torch.zeros((len(detections)))
        total_true_masks = len(ground_truths)
    for detection_idx, detection in enumerate(detections):
        ground_truth_img = [bbox for bbox in ground_truths if bbox[0] == detection[0]]
        num_gts = len(ground_truth_img)
        best_iou = 0
        detection_post = torch.nn.functional.interpolate(detection, size=(1250, 1250), scale_factor=None, mode='nearest', align_corners=None, recompute_scale_factor=None)
        for idx, gt in enumerate(ground_truth_img):
            iou = torch.sum((torch.mul(detection_post[2], gt[2]))[2])/(torch.sum(detection_post[2]) + torch.sum(gt[2]) - torch.mul(detection_post[2], gt[2]))
            if iou > best_iou:
                best_iou = iou
                best_gt_idx = idx
        if best_iou > iou_threshold:
            if amount_masks[detection[0]][best_gt_idx] == 0:
                TP[detection_idx] = 1
                amount_masks[detection[0]][best_gt_idx] = 1
            else:
                FP[detection_idx] = 1
        else:
```

```
            FP[detection_idx] = 1
    # [1, 1, 0, 1, 0] -> [1, 2, 2, 3, 3]
    TP_cumsum = torch.cumsum(TP, dim = 0)
    FP_cumsum = torch.cumsum(FP, dim = 0)
    recalls = TP_cumsum / (total_true_masks + epsilon)
    precisions = torch.divide(TP_cumsum, (TP_cumsum + FP_cumsum + epsilon))
    precisions = torch.cat((torch.tensor([1]), precisions))
    recalls = torch.cat((torch.tensor([0]), recalls))
    average_precision = torch.trapz(precisions, recalls)

    return average_precision


def mean_average_precision_15(pred_masks, true_masks, iou_threshold = 0.5, num_classes = 1):
    average_precision = 0
    epsilon = 1e-6

    detections = []
    ground_truths = []
    flags = []
    for index, detection in enumerate(pred_masks):
        if torch.sum(detection[2]) > 25:
            detections.append(detection)
            flags.append(index)
    for flag in flags:
        ground_truths.append(true_masks[flag])
    #img 0 has 3 masks
    #img 1 has 5 masks
    #amount_masks = {0:3, 1:5}
    amount_masks = Counter([gt[0] for gt in ground_truths])
    for key, val in amount_masks.items():
        #amount_boxes = {0:torch.tensor([0,0,0]), 1:torch.tensor([0,0,0,0,0])}
        amount_masks[key] = torch.zeros(val)
        detections.sort(key = lambda x: x[1], reverse = True)
        TP = torch.zeros((len(detections)))
        FP = torch.zeros((len(detections)))
        total_true_masks = len(ground_truths)
    for detection_idx, detection in enumerate(detections):
        ground_truth_img = [bbox for bbox in ground_truths if bbox[0] == detection[0]]
        num_gts = len(ground_truth_img)
        best_iou = 0
        detection_post = torch.nn.functional.interpolate(detection, size=(1250, 1250), scale_factor=None, mode='nearest', align_corners=None, recompute_scale_factor=None)
        for idx, gt in enumerate(ground_truth_img):
            iou = torch.sum((torch.mul(detection_post[2], gt[2]))[2])/(torch.sum(detection_post[2]) + torch.sum(gt[2]) - torch.mul(detection_post[2], gt[2]))
            if iou > best_iou:
                best_iou = iou
                best_gt_idx = idx
        if best_iou > iou_threshold:
            if amount_masks[detection[0]][best_gt_idx] == 0:
                TP[detection_idx] = 1
                amount_masks[detection[0]][best_gt_idx] = 1
            else:
                FP[detection_idx] = 1
        else:
            FP[detection_idx] = 1
    # [1, 1, 0, 1, 0] -> [1, 2, 2, 3, 3]
    TP_cumsum = torch.cumsum(TP, dim = 0)
    FP_cumsum = torch.cumsum(FP, dim = 0)
    recalls = TP_cumsum / (total_true_masks + epsilon)
    precisions = torch.divide(TP_cumsum, (TP_cumsum + FP_cumsum + epsilon))
```

```
    precisions = torch.cat((torch.tensor([1]), precisions))
    recalls = torch.cat((torch.tensor([0]), recalls))
    average_precision = torch.trapz(precisions, recalls)

    return average_precision

def mean_average_precision_50(pred_masks, true_masks, iou_threshold = 0.5, num_classes = 1):
    average_precision = 0
    epsilon = 1e-6

    detections = []
    ground_truths = []
    flags = []
    for index, detection in enumerate(pred_masks):
        if torch.sum(detection[2]) > 50:
            detections.append(detection)
            flags.append(index)
    for flag in flags:
        ground_truths.append(true_masks[flag])
    #img 0 has 3 masks
    #img 1 has 5 masks
    #amount_masks = {0:3, 1:5}
    amount_masks = Counter([gt[0] for gt in ground_truths])
    for key, val in amount_masks.items():
        #amount_boxes = {0:torch.tensor([0,0,0]), 1:torch.tensor([0,0,0,0,0])}
        amount_masks[key] = torch.zeros(val)
        detections.sort(key = lambda x: x[1], reverse = True)
        TP = torch.zeros((len(detections)))
        FP = torch.zeros((len(detections)))
        total_true_masks = len(ground_truths)
    for detection_idx, detection in enumerate(detections):
        ground_truth_img = [bbox for bbox in ground_truths if bbox[0] == detection[0]]
        num_gts = len(ground_truth_img)
        best_iou = 0
        detection_post = torch.nn.functional.interpolate(detection, size=(1250, 1250), scale_factor=None, mode='nearest', align_corners=None, recompute_scale_factor=None)
        for idx, gt in enumerate(ground_truth_img):
            iou = torch.sum((torch.mul(detection_post[2], gt[2]))[2])/(torch.sum(detection_post[2]) + torch.sum(gt[2]) - torch.mul(detection_post[2], gt[2]))
            if iou > best_iou:
                best_iou = iou
                best_gt_idx = idx
        if best_iou > iou_threshold:
            if amount_masks[detection[0]][best_gt_idx] == 0:
                TP[detection_idx] = 1
                amount_masks[detection[0]][best_gt_idx] = 1
            else:
                FP[detection_idx] = 1
        else:
            FP[detection_idx] = 1
    # [1, 1, 0, 1, 0] -> [1, 2, 2, 3, 3]
    TP_cumsum = torch.cumsum(TP, dim = 0)
    FP_cumsum = torch.cumsum(FP, dim = 0)
    recalls = TP_cumsum / (total_true_masks + epsilon)
    precisions = torch.divide(TP_cumsum, (TP_cumsum + FP_cumsum + epsilon))
    precisions = torch.cat((torch.tensor([1]), precisions))
    recalls = torch.cat((torch.tensor([0]), recalls))
    average_precision = torch.trapz(precisions, recalls)

    return average_precision
```

Post Processing

solo_visualization

December 17, 2021

[ ]:
```
!gpustat
```

[ ]:
```
import os
os.environ["CUDA_VISIBLE_DEVICES"] = "6"
```

[4]:
```
from mpl_toolkits.mplot3d import Axes3D
from sklearn.cluster import DBSCAN
from tqdm import tqdm, trange
import random
import numpy as np
import torch
import torchvision
from torchvision import transforms
import torch.nn.functional as F
from torch import nn
from torch.utils.data import Dataset, DataLoader
from torchvision.models.detection import maskrcnn_resnet50_fpn
import h5py
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import matplotlib.patches as patches
from PIL import Image
import os
from IPython.display import clear_output
import math
import matplotlib.pyplot as plt
import PIL
import timeit

import warnings
warnings.filterwarnings("ignore")
```

[5]:
```
draw_box = lambda x1, y1, x2, y2: np.array([[x1, y1], [x1, y2], [x2, y2], [x2,␣
↪y1], [x1, y1]])
```

1 Load Data

[6]:
```
from data_loader import *
DATA_DIR = "../processdata/"
test_files = list(np.load("test_filenames.npy"))
train_files = list(np.load("train_filenames.npy"))
batch_size = 1
transforms = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
                                             torchvision.transforms.
 ↪Grayscale(),
                                             torchvision.transforms.
 ↪Normalize([0.5], [0.5])])
train_loader = data_loader(train_files, DATA_DIR, batch_size =␣
 ↪batch_size,transforms = transforms)
test_loader = data_loader(test_files, DATA_DIR, batch_size =␣
 ↪batch_size,transforms = transforms)
```

1.1 Visual Inspection

[7]:
```
# data = next(iter(train_loader))
for data in train_loader:
    break
images, masks, bboxes, batch = data
```

2 Model

[43]:
```
# loading existing configuration
from architecture import *
# from config import *
checkpoint_path = f'../check_points/21_12_15_3'
hyper_parameters = np.load(f'{checkpoint_path}/hyper_parameters.npy',␣
 ↪allow_pickle = True).item()
print(hyper_parameters)
num_epoch = 75
eps = hyper_parameters['eps']

num_grids = hyper_parameters['num_grids']
scale_ranges = hyper_parameters['scale_ranges']
postprocess_cfg = dict()
```

```
{'num_grids': [90, 60, 30, 10], 'scale_ranges': ((0.05, 0.4), (0.1, 0.5), (0.4,
0.7), (0.6, 1)), 'eps': 0.2, 'optimizer': 'Adam', 'mask_weight': 1.0,
'cate_weight': 1.0}
```

[44]:
```
print(hyper_parameters)
```

```
{'num_grids': [90, 60, 30, 10], 'scale_ranges': ((0.05, 0.4), (0.1, 0.5), (0.4,
0.7), (0.6, 1)), 'eps': 0.2, 'optimizer': 'Adam', 'mask_weight': 1.0,
'cate_weight': 1.0}
```

2.1 SOLO

[45]:
```
class SOLO(SOLO_head):
    def __init__(self,
                 scale_ranges=scale_ranges,
                 num_grids=num_grids,
                 mask_loss_cfg=dict(weight=3),
                 cate_loss_cfg=dict(gamma=2,
                                    alpha=0.25,
                                    weight=1),
                 postprocess_cfg=postprocess_cfg):
        super(SOLO, self).__init__(scale_ranges, num_grids, mask_loss_cfg,␣
↪cate_loss_cfg, postprocess_cfg)


        self.backbone = FPN(Bottleneck, [3,4,6,3])

    def forward(self, images, device, eval=False, original_img = None):

        feature_pyramid = list(self.backbone(images))

        category_predictions = []
        mask_predictions = []

        if eval == True:

            for i, v in enumerate(feature_pyramid):
                num_grid = self.num_grids[i]
                category_pred = nn.functional.interpolate(v, size=[num_grid,␣
↪num_grid], mode="bilinear", align_corners = True)
                category_pred = self.cate_branch(category_pred)
                category_predictions.append(self.points_nms(category_pred).
↪permute(0,2,3,1))

                meshxy = torch.stack(torch.meshgrid(torch.linspace(-1,1,v.
↪shape[-2]), torch.linspace(-1,1,v.shape[-1])))
                batch_meshxy = torch.tile(meshxy, (images.shape[0],1,1,1)).
↪to(device)
                new_v = torch.cat((v, batch_meshxy), 1)

                mask_pred = self.ins_outs[i](self.ins_branch(new_v))
```

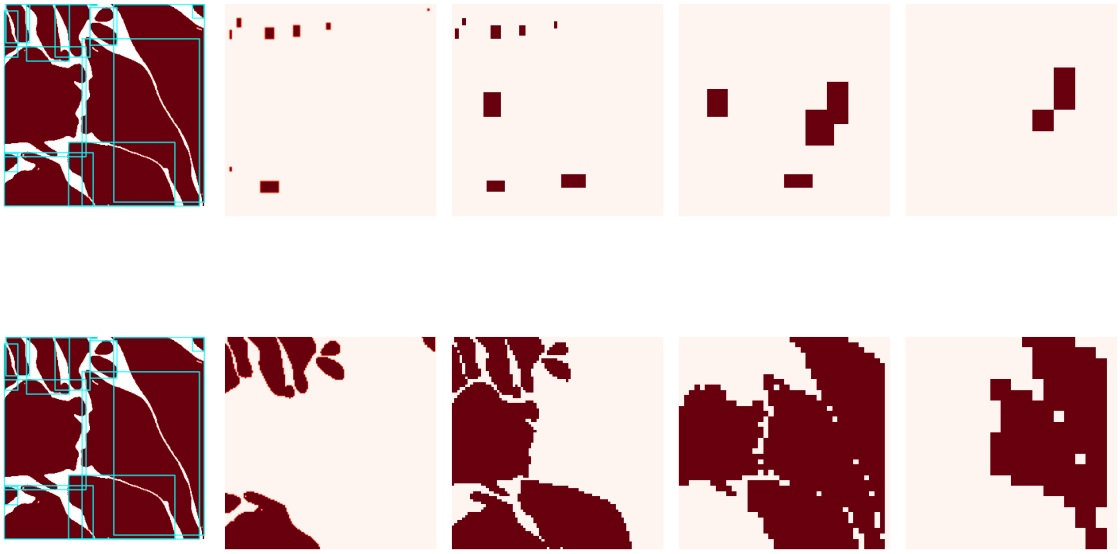
```
                mask_predictions.append(nn.functional.interpolate(mask_pred,␣
↪size=original_img, mode='bilinear', align_corners = True))
            return category_predictions, mask_predictions

        else:
            for i, v in enumerate(feature_pyramid):
                num_grid = self.num_grids[i]
                category_pred = nn.functional.interpolate(v, size=[num_grid,␣
↪num_grid], mode="bilinear", align_corners = True)
                category_predictions.append(self.cate_branch(category_pred))

                meshxy = torch.stack(torch.meshgrid(torch.linspace(-1,1,v.
↪shape[-2]), torch.linspace(-1,1,v.shape[-1])))
                batch_meshxy = torch.tile(meshxy, (images.shape[0],1,1,1)).
↪to(device)
                new_v = torch.cat((v, batch_meshxy), 1)

                mask_pred = self.ins_outs[i](self.ins_branch(new_v))

                mask_predictions.append(nn.functional.interpolate(mask_pred,␣
↪scale_factor=2, mode='bilinear', align_corners = True))
            return category_predictions, mask_predictions
```

[46]:
```
# device = 'cuda:0'
device = 'cuda:0'
model = SOLO().to(device)
backbone_out = model.backbone(torch.zeros(0,1,313,313).to(device))
model.featmap_sizes = [list(f.shape[-2:]) for f in backbone_out]
labels = [torch.ones(b.shape[0]) for b in bboxes]
masks, bboxes = [torch.Tensor(m) for m in masks], [torch.Tensor(b) for b in␣
↪bboxes]
target = model.generate_single_target(bboxes[0], labels[0], masks[0], model.
↪featmap_sizes, eps = eps)
```

```
((0.05, 0.4), (0.1, 0.5), (0.4, 0.7), (0.6, 1))
[90, 60, 30, 10]
```

[47]:
```
for data in train_loader:
    break
images, masks, bboxes, batch = data
labels = [torch.ones(b.shape[0]) for b in bboxes]
masks, bboxes = [torch.Tensor(m) for m in masks], [torch.Tensor(b) for b in␣
↪bboxes]
target = model.generate_single_target(bboxes[0], labels[0], masks[0], model.
↪featmap_sizes, eps = eps)
```

```
plt.figure(figsize=(20, 4))

plt.subplot(1,5,1)
for i, (msk, box) in enumerate(zip(masks[0], bboxes[0])):

    plt.imshow(msk, cmap = 'Reds', alpha = msk * 1.)
    plt.plot(*draw_box(*box * 313).T, color = 'aqua')
plt.axis('off')

for i in range(4):


    plt.subplot(1,5,i + 2)
    plt.imshow(target[1][i].reshape(num_grids[i],num_grids[i]), cmap = 'Reds')
    plt.axis('off')


plt.tight_layout()
plt.savefig('pixel_activation.pdf', dpi = 100)
plt.show()
plt.close()

plt.figure(figsize=(20, 4))

plt.subplot(1,5,1)
for i, (msk, box) in enumerate(zip(masks[0], bboxes[0])):

    plt.imshow(msk, cmap = 'Reds', alpha = msk * 1.)
    plt.plot(*draw_box(*box * 313).T, color = 'aqua')
plt.axis('off')

for i in range(4):


    plt.subplot(1,5,i + 2)
    plt.imshow(target[0][i].sum(0).reshape(*[ f * 2 for f in model.
 ↪featmap_sizes[i]]) > 0, cmap = 'Reds')
    plt.axis('off')

plt.tight_layout()
plt.savefig('mask_assignment.pdf', dpi = 100)
plt.show()
plt.close()
```



3 Loss plot

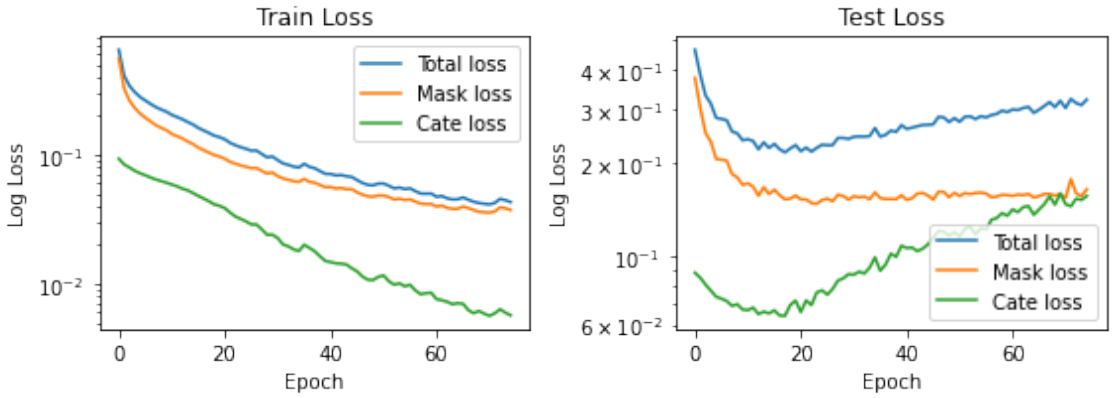
[48]:
```
model.load_state_dict(torch.load(f'{checkpoint_path}/solo-epoch-{num_epoch}.
↪pth')['model_state_dict'])
```

[48]:
```
<All keys matched successfully>
```

[49]:
```
loss_train_stor = np.load(f'{checkpoint_path}/loss-train-epoch-{num_epoch}.npy')
loss_val_stor = np.load(f'{checkpoint_path}/loss-val-epoch-{num_epoch}.npy')
```

[50]:
```
train_loss_avg = np.array(loss_train_stor).reshape(num_epoch,-1,3).mean(1)
test_loss_avg = np.array(loss_val_stor).reshape(num_epoch,-1,3).mean(1)

plt.figure(figsize = (8,3))
plt.subplot(1,2,1)
plt.plot(np.linspace(0, num_epoch, train_loss_avg.shape[0], endpoint = False),␣
↪train_loss_avg[:,2], label = 'Total loss')
plt.plot(np.linspace(0, num_epoch, train_loss_avg.shape[0], endpoint = False),␣
↪train_loss_avg[:,1], label = 'Mask loss')
plt.plot(np.linspace(0, num_epoch, train_loss_avg.shape[0], endpoint = False),␣
↪train_loss_avg[:,0], label = 'Cate loss')
plt.yscale('log')
plt.xlabel('Epoch')
plt.ylabel('Log Loss')
plt.title('Train Loss')
plt.legend()
plt.tight_layout()
```

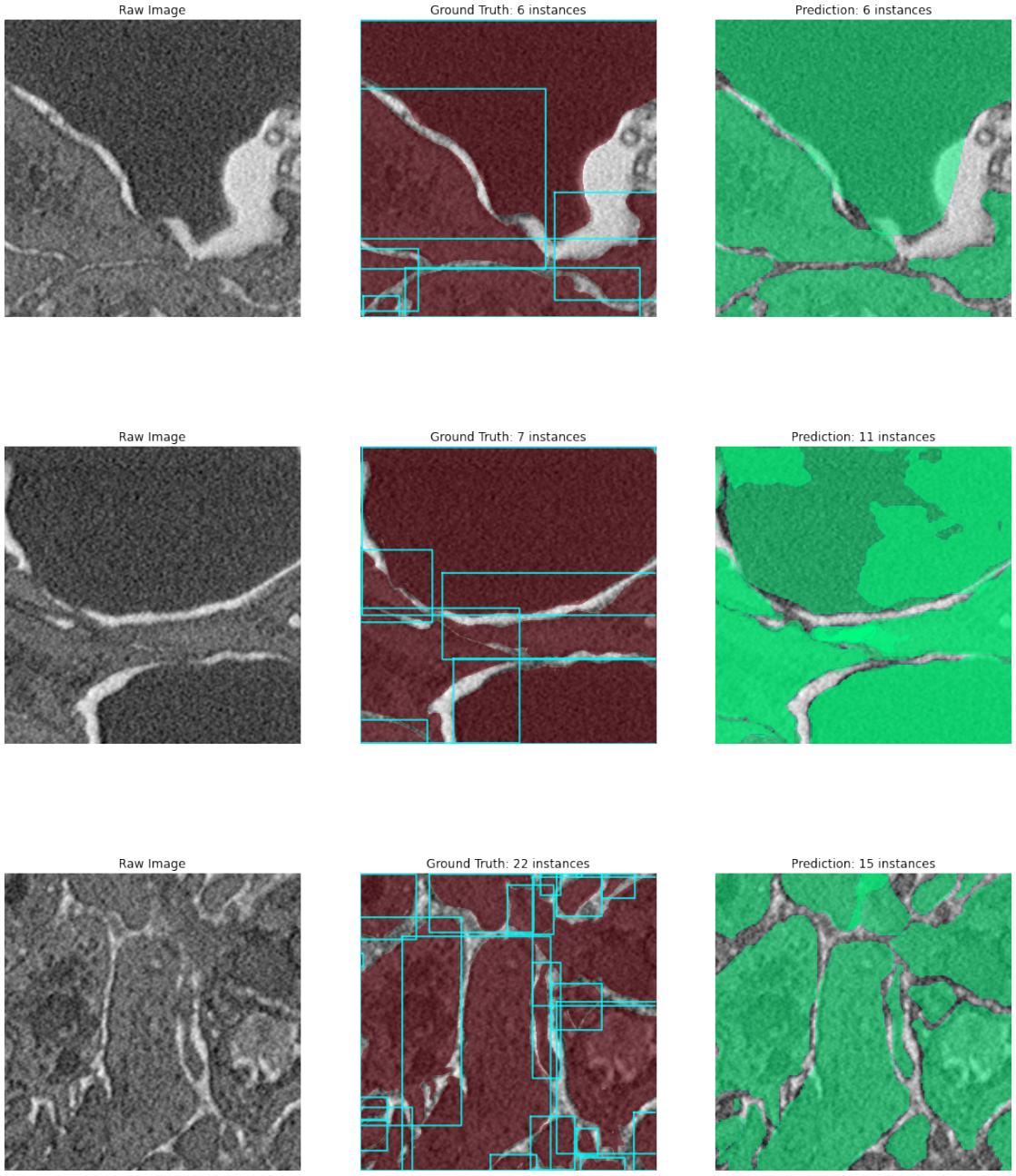
```
# plt.savefig(f'{checkpoint_path}/loss_plot.pdf', dpi = 100)
plt.subplot(1,2,2)
plt.plot(np.linspace(0, num_epoch, test_loss_avg.shape[0], endpoint = False),␣
 ↪test_loss_avg[:,2], label = 'Total loss')
plt.plot(np.linspace(0, num_epoch, test_loss_avg.shape[0], endpoint = False),␣
 ↪test_loss_avg[:,1], label = 'Mask loss')
plt.plot(np.linspace(0, num_epoch, test_loss_avg.shape[0], endpoint = False),␣
 ↪test_loss_avg[:,0], label = 'Cate loss')
plt.yscale('log')
plt.xlabel('Epoch')
plt.ylabel('Log Loss')
plt.title('Test Loss')
plt.legend()
plt.tight_layout()
plt.savefig(f'{checkpoint_path}/solo-epoch-{num_epoch}-train-losses.pdf', dpi =␣
 ↪100)
plt.show()
plt.close()
```

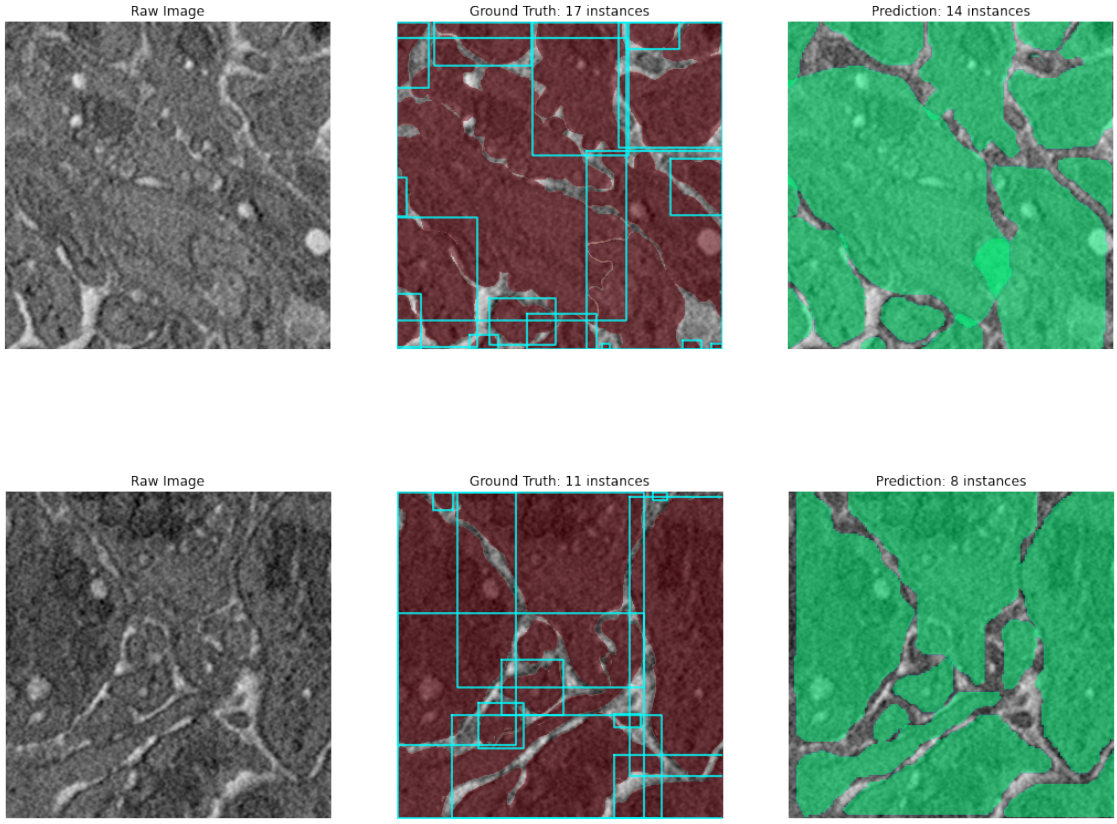


[51]:
```
postprocess_cfg = dict(cate_thresh=0.5,
                       ins_thresh=0.5,
                       pre_NMS_num=400,
                       # pre_NMS_num=50,
                       keep_instance=15)
nms_score_threshold = postprocess_cfg['cate_thresh'] * 0.95

model.postprocess_cfg = postprocess_cfg
```

[52]:
```
batch = 0
n_show = 5

# predict
```

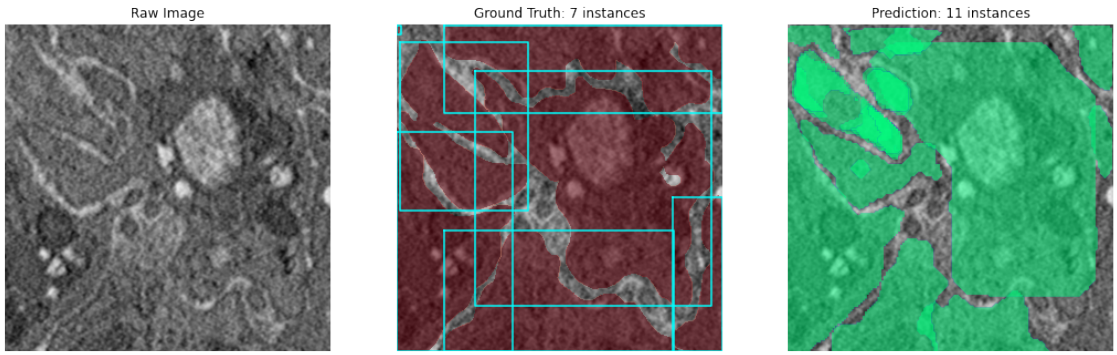
```
model.eval()
for i, data in enumerate(train_loader):
# for i, data in enumerate(test_loader):

    images, masks, bboxes, _ = data
    masks, bboxes = [torch.Tensor(m) for m in masks], [torch.Tensor(b) for b in␣
↪bboxes]
    original_img= [int(313/2),int(313/2)]

    post_out = model.post_processing(*model.forward(images.to(device),
                                                    device=device,
                                                    eval=True,
                                                    original_img=original_img),
                                     nms_score_threshold = nms_score_threshold)

    plt.figure(figsize = (18,6))

    plt.subplot(1, 3, 1)
    plt.title('Raw Image')
    plt.imshow(images[batch].squeeze(), cmap = 'gray')
    plt.axis('off')

    plt.subplot(1,3,2)
    plt.title(f'Ground Truth: {len(bboxes[batch])} instances')
    plt.imshow(images[batch].squeeze(), cmap = 'gray')
    for box, msk in zip(bboxes[batch], masks[batch]):
        plt.plot(*draw_box(*box * 313).T, color = 'aqua')
        plt.imshow(msk, alpha = msk * 0.5, cmap = 'Reds')
    plt.axis('off')

    plt.subplot(1,3,3)
    plt.title(f'Prediction: {len(post_out[batch])} instances')
    plt.imshow(images[batch].squeeze(), cmap = 'gray')
    for msk in post_out[0]:
        m = (nn.functional.interpolate(msk[None, None].float(), size=images.
↪shape[-2:], mode='bilinear') > 0.5)[0,0].int().cpu().numpy()
        plt.imshow(m, alpha = m * 0.5, cmap = 'winter')
    plt.axis('off')

    plt.savefig(f'{checkpoint_path}/solo-epoch-{num_epoch}-train-results-{i}.
↪pdf', dpi = 100)
    plt.show()
    plt.close()
    if i == n_show-1:
        break
```

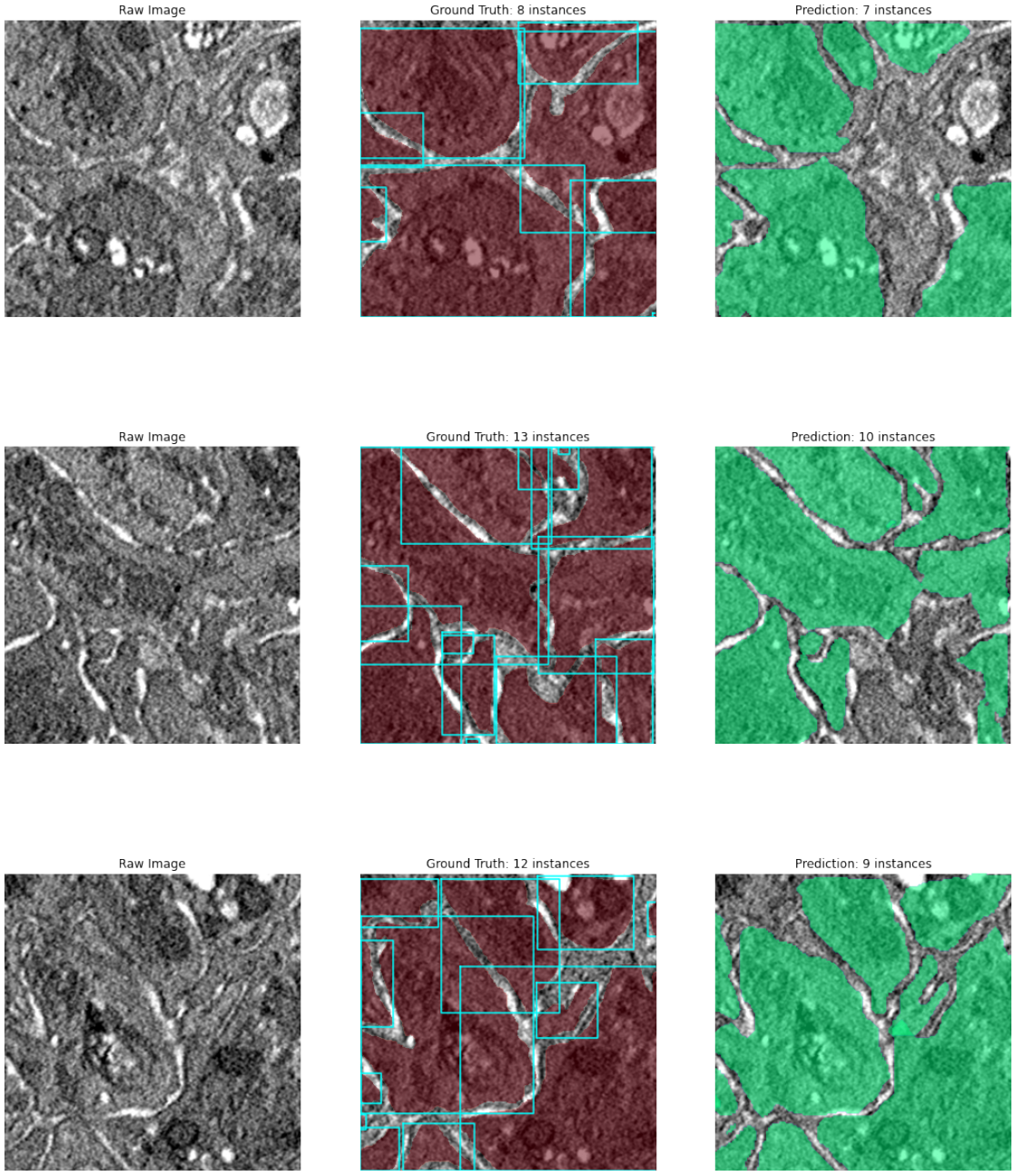

Raw Image
Ground Truth: 6 instances
Prediction: 6 instances
Raw Image
Ground Truth: 7 instances
Prediction: 11 instances
Raw Image
Ground Truth: 22 instances
Prediction: 15 instances

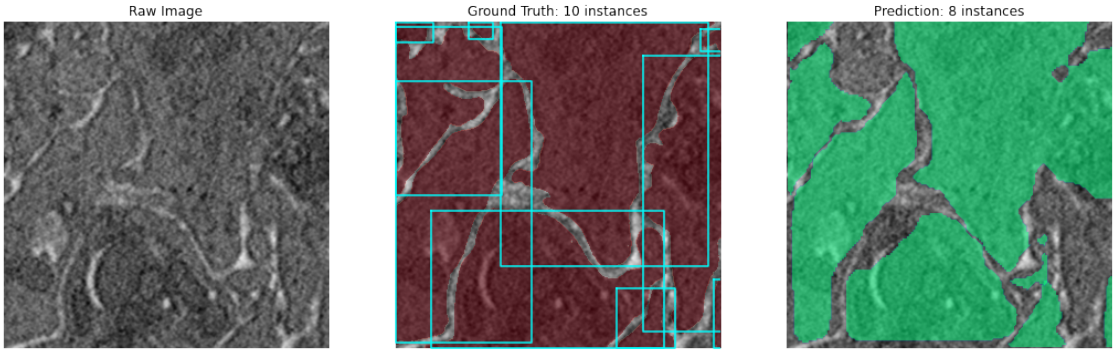


[53]:
```
# predict
model.eval()
# for i, data in enumerate(train_loader):
for i, data in enumerate(test_loader):

    images, masks, bboxes, _ = data
    masks, bboxes = [torch.Tensor(m) for m in masks], [torch.Tensor(b) for b in␣
 ↪bboxes]
    original_img= [int(313/2),int(313/2)]

    post_out = model.post_processing(*model.forward(images.to(device),
                                                    device=device,
                                                    eval=True,
                                                    original_img=original_img),
                                     nms_score_threshold = nms_score_threshold)

    plt.figure(figsize = (18,6))

    plt.subplot(1, 3, 1)
    plt.title('Raw Image')
    plt.imshow(images[batch].squeeze(), cmap = 'gray')
```

```
    plt.axis('off')

    plt.subplot(1,3,2)
    plt.title(f'Ground Truth: {len(bboxes[batch])} instances')
    plt.imshow(images[batch].squeeze(), cmap = 'gray')
    for box, msk in zip(bboxes[batch], masks[batch]):
        plt.plot(*draw_box(*box * 313).T, color = 'aqua')
        plt.imshow(msk, alpha = msk * 0.5, cmap = 'Reds')
    plt.axis('off')

    plt.subplot(1,3,3)
    plt.title(f'Prediction: {len(post_out[batch])} instances')
    plt.imshow(images[batch].squeeze(), cmap = 'gray')
    for msk in post_out[0]:
        m = (nn.functional.interpolate(msk[None, None].float(), size=images.
↪shape[-2:], mode='bilinear') > 0.5)[0,0].int().cpu().numpy()
        plt.imshow(m, alpha = m * 0.5, cmap = 'winter')
    plt.axis('off')

    plt.savefig(f'{checkpoint_path}/solo-epoch-{num_epoch}-test-results-{i}.
↪pdf', dpi = 100)
    plt.show()
    plt.close()
    if i == n_show-1:
        break
```




Raw Image
Ground Truth: 8 instances
Prediction: 7 instances
Raw Image
Ground Truth: 13 instances
Prediction: 10 instances
Raw Image
Ground Truth: 12 instances
Prediction: 9 instances



[54]:
```
batch_size = 4
transforms = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
                                             torchvision.transforms.
 ↪Grayscale(),
                                             torchvision.transforms.
 ↪Normalize([0.5], [0.5])])
train_loader = data_loader(train_files, DATA_DIR, batch_size =␣
 ↪batch_size,transforms = transforms)
test_loader = data_loader(test_files, DATA_DIR, batch_size =␣
 ↪batch_size,transforms = transforms)
%timeit out = model.forward(images.
 ↪to(device),device=device,eval=True,original_img=original_img)
%timeit post_out = model.post_processing(*model.forward(images.
 ↪to(device),device=device,eval=True,original_img=original_img),nms_score_threshold␣
 ↪= nms_score_threshold)
```

```
68 ms ± 3.24 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
309 ms ± 6.77 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```