Master's Thesis

Semantic Segmentation And Object Tracking For Elongated Macrophage Cells

---

Hakan Sarp Aydemir

Albert-Ludwigs-University Freiburg
Faculty of Engineering
Department of Computer Science
Chair for Pattern Recognition and Image Processing

November 28th, 2022

Writing period

01. 04. 2022 – 28. 10. 2022

Examiners

Prof. Dr. Thomas Brox

Prof. Dr.-Ing. Matthias Teschner

Adviser

Yassine Marrakchi

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

Abstract

Image segmentation has been one of the most researched and most successful application areas of artificial intelligence and deep learning. Instance segmentation, in particular, has proven to be a more complex problem compared to semantic segmentation. The most established techniques to the instance segmentation method use bounding boxes, rectangular boxes that enclose the objects of interest in the image, as their basis before solving the pixel-wise segmentation problem. This technique proves to be not very effective in case of intersecting objects of interest with elongated shapes. This work investigates this problem by implementing the DiskMask algorithm, an algorithm that gets rid of the bounding boxes, in PyTorch to enable the future research on this problem domain to be done more conveniently through Python. The results are evaluated on C. Elegans, and compared with the published results of the original DiskMask paper.

Contents

List of Figures

1 Introduction

From system-critical use-cases such as self-driving cars to complex research problems in different domains such as medicine, distinguishing between different instances of the same class of objects is a very useful information to have. Instance segmentation has been a very important and heavily researched problem in computer vision because of this reason. Because of the many different use-cases of instance segmentation in various domains, it is difficult to come up with a generalized method that performs well in all of these use-cases and domains.

One of the most established methods to approach the problem of instance segmentation is Mask-RCNN [1]. As a middle step, this method uses bounding boxes that enclose objects of interest in order to solve a detection problem. Although this popular method works well in most cases with optimized hyper-parameters, it leaves room for improvement in instances of elongated and intersecting objects of interest. Bounding boxes are intuitively not a good representation for elongated objects because of some objects' organic and non-geometric shapes contrasting the rectangular shape of the bounding boxes, which makes it difficult to represent the objects' spatial information. An example of such elongated objects is shown in Figure 1, which is a microscopy image of several C. Elegans roundworms taken from the C. Elegans dataset [2]. An example of an image can be seen in 1.

Because of these reasons, solutions that do not use bounding boxes are more likely to be better fits for this kind of problem. DiskMask [3] gets rid of the bounding boxes by creating a feature map where the information related to the objects, such as their location, is collected into disks and the segmentation maps are decoded from these disks using a U-Net [4].

In this work, we investigate this problem of segmentation of elongated objects by implementing the DiskMask algorithm, which is originally coded in Matlab and uses Caffe [5], in PyTorch [6] and testing it on C. Elegans dataset. We perform instance segmentation and report the results while creating a tool that can utilize the advantages and extensibility of PyTorch for future segmentation research for elongated objects.

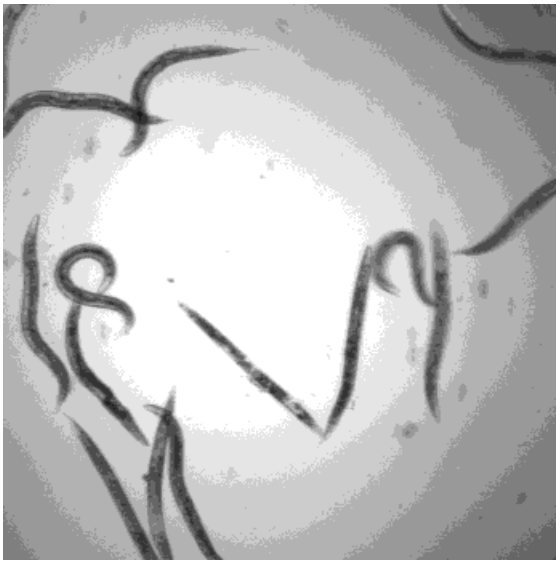


Figure 1: Example of elongated roundworms. Elongated objects make it difficult to have an accurate bounding box representation: Roundworms, C. Elegans.

2 Related Work

This chapter establishes the background information to understand the ideas and reasonings behind the implementation details of the DiskMask PyTorch program. It is divided into three sections. First section presents the more established and traditional instance segmentation methods and techniques, which helps the reader to understand the differences between the DiskMask methods. Second section establishes a core component, U-Net [4], that is used in the DiskMask paper to create the feature map and the segmentation masks. Finally, the third section presents the DiskMask paper in detail to create a better understanding between the original DiskMask paper and the PyTorch implementation.

2.1 Instance Segmentation

Instance segmentation is one of the heavily researched areas of computer vision. Instance segmentation provides pixel-wise segmentation of objects to different classes while separating distinct objects of the same class. In a sense, it combines object detection and semantic segmentation.

In this section we give background information about the traditional approaches and what is being replaced in DiskMask.

2.1.1 R-CNN

R-CNN [7] works by extracting region proposals from the image. Then the convolutional neural network (CNN) extracts the features for each of the region proposals. Finally, linear support vector machine (SVM) classifies the input regions into different classes. In the end, we have the object's class and location of the bounding box.

2.1.2 Mask R-CNN

Mask R-CNN [1] is one of the most popular approaches to instance segmentation. It extends the ideas from R-CNN and uses a region proposal network (RPN) [8],

a fully convolutional network, to propose possible object bounding boxes. Then, Mask R-CNN produces a binary segmentation mask for each object in addition to predicting the class and the bounding box location like its predecessors.

The dependence of this method to bounding boxes is not a very effective way of representing the elongated objects for the reasons stated in Section 1.

2.2 U-Net

U-Net [4] is a very important building block of the DiskMask model architecture and an overlap-tile strategy is used for inference in the DiskMask paper. U-Net is used for pixel-wise semantic segmentation and the architecture consists of two main parts, the encoder network followed by the decoder network. The encoder network has a contracting path of convolutions followed by downsampling operations such as max pooling. This part encodes the input image to feature maps for different resolutions. The decoder network has an expansive path of convolutions followed by upsampling operations. This part projects the features from the contracting path into higher resolution.

2.2.1 Skip Connections

The information from different layers of the contracting path is copied and cropped to be concatenated with their respective upsampled layers on the expansive path. This makes use of the high resolution features from the contracting path for the convolution layers in the expansive path to create a more precise output. These skip connections are also used in the DiskMask architecture. The skip connections can be seen as gray arrows in 2.

2.2.2 Overlap-Tile Strategy

This strategy from the U-Net paper allows us to do the segmentation operation on arbitrarily large images. The original image is separated into different tiles where the full context is available in the input image. In U-Net paper, the missing context on the border tiles are provides by mirroring the tile to extrapolate information. Diskmask uses this strategy to deal with border tiles and multiple tiles with the same object in them.

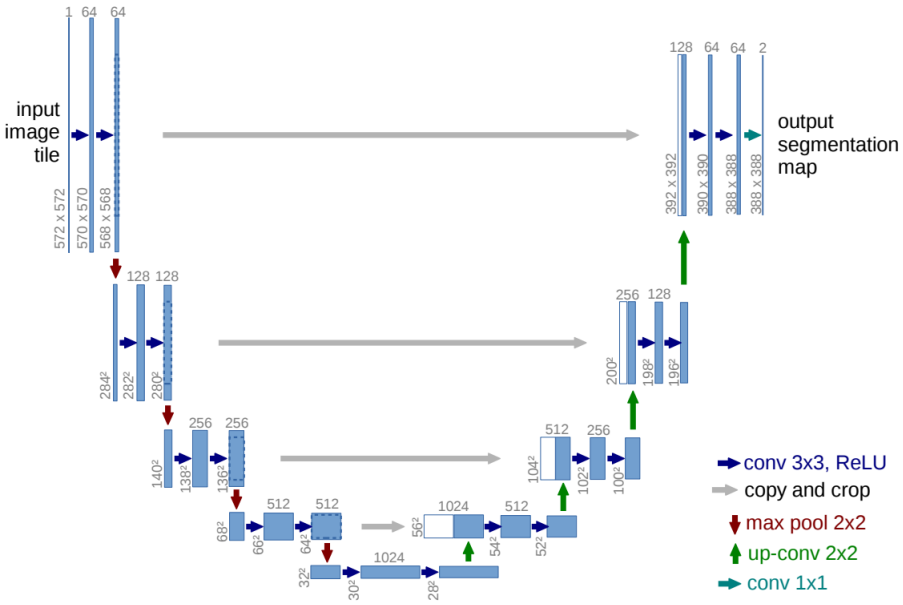


Figure 2: U-Net architecture. The skip connections are represented as gray arrows pointed from the contracting path of the encoder to the respective convolution blocks on the expansive path of the decoder

2.2.3 Weighted Soft-max Loss

A weight map is calculated for every pixel to compentsate for the difference in frequency of pixels in certain classes, such as foreground-background class imbalance. The formula below is used to calculate the weighted cross-entropy loss where $w(x)$ is the weight map and the p_l is the soft-max.

$$E = \sum_{x \in \Omega} w(x) \log(p_{l(x)}(x)) \tag{1}$$

In short, the resulting loss is multiplied with the weight map value at each pixel.

2.3 DiskMask

DiskMask is an instance segmentation method that is specifically effective in elongated and overlapping objects. It achieves this by getting rid of the popular Mask R-CNN approach of bounding boxes, which is"...an artificial construct and suboptimal for elongated objects" [3].

2.3.1 Gateways and Disks

The bounding box representation is replaced by feature representations on a regular grid. The information regarding the detected objects are collected in local disks. These disks are also called "gateways" and segmentation masks are decoded from these gateways instead of bounding boxes. An object ordering can be derived from these gateways since they are not touching and this way we can get information about which segmentation mask belongs to which gateway, helping us set apart the different instances of overlapping objects. The reference point of an object $\mathbf{p}_k$ [9] is an identifier of the object's location and can be placed at any place as long as it's position relative to the object does not change. Every one of these reference points is coupled with disks of different radiuses $\mathbf{r}_k$ and the radiuses of the ground truth disks shrink when two object are close or overlapping so their size is adaptive. This prevents the disks themselves from overlapping so we can have different segmentation masks for different objects. These disks are used to mitigate the class imbalance problem of having sparse positions, focusing the network on the local region of the object.

2.3.2 Network Architecture

DiskMask architecture consists of two encoder-decoder networks as in Section 2.2. The entire compositional network can be seen in 3

The second encoder-decoder network, named the DecoNet, is a modified U-Net with half the number of channels and the first convolution block replaced with a single 5 x 5 convolution operation with stride 2 instead of two 3 x 3 convolutions + a max-pooling operation. The pixel-wise sigmoid cross entropy loss is calculated to train the DecoNet. The input of this DecoNet network is a feature map F of dimension 324x324x64 with activated object features and the output is the 132x132x1 segmentation map. In short, DecoNet is used to derive a segmentation mask for a specific object instance.

The first enoder-decoder network, named the EncoNet, is also of the same network architrecture but without the intra-net skip connections as in fig 2, forcing the signal to pass throguh all the layers. It takes the image crop as an input and gives two tensors with channel sizes 64 and 2 as an output. The first head of the network defines the dimensionality of the feature map F while the second head maps the output to disk logits. The weighted soft-max cross entropy loss is calculated on the

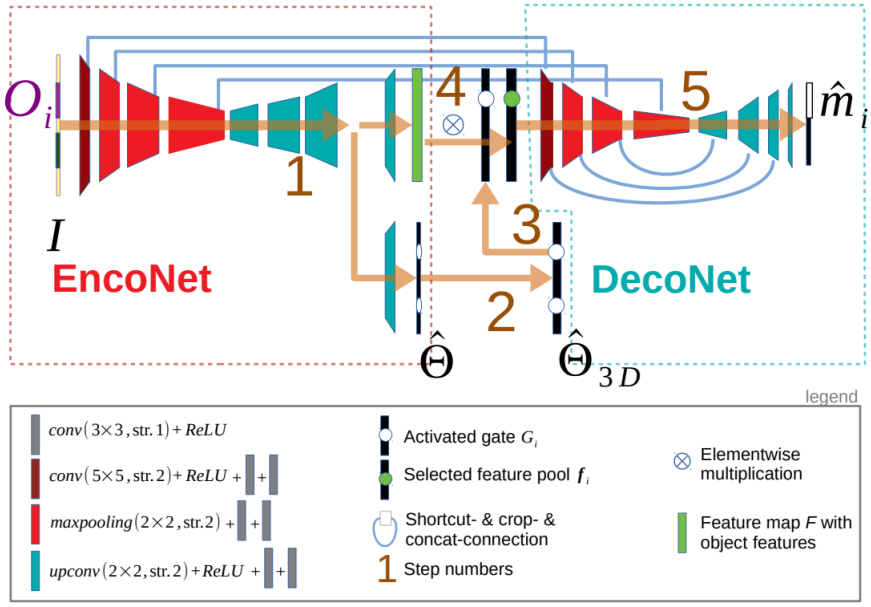


Figure 3: Diskmask architecture. Diskmask architecture is made of two encoder-decoder networks, EncoNet and DecoNet. The shortcut+crop+concatenate operations are represented by the intra-net and inter-net connections. Step numbers for a forward pass through the network are given. Activation of gateways and local pools are also represented in the figure.

second head's output [4]. In short, EncoNet takes the image as input and predicts the gateway locations and encodes object information.

In addition to intra-net connections as in U-Net, there are also inter-net connections in DiskMask which can be seen in fig 3. These skip connections between EncoNet and DecoNet take the feature maps from the end of each convolution block at the contracting path of EncoNet. These feature maps are then copied, cropped and concatenated to the beginning of the respective convolution blocks in the contracting path of DecoNet.

2.3.3 Training and Inference

During inference, the image input is taken by the EncoNet and it outputs a feature map F with object features and disk logits $\hat{\Theta}$. The connected components from $\hat{\Theta}$ are used to extract disk positions and number of objects. $\hat{\Theta}$ is then replicated along its channel dimension to produce the 3D segmentation masks for different disks. A single

gateway G_i among these disks is activated by zeroing out the others. The objects local pool f_i is acquired by multiplying the feature map F and the 3D segmentation masks. Then the DecoNet takes these as input and creates the segmentation masks using f_i and the inter-net skip connections. The overlap-tile strategy from Section 2.2 is used to do inference.

During the training, the entire compositional network is trained end-to-end. The loss functions mentioned in Section 2.3.2 are used for EncoNet and DecoNet separately with the same weight on both losses. As a difference from the inference, a subset of the gateways are opened with a 50% chance of opening instead of only one of them. The objects' gateways that are on the borders of DecoNet's input are ignored. The radius of the disks are randomly shrunk in order to prevent overfitting to a certain size.

3 Implementation Details

To tackle the problem of instance segmentation of elongated objects, we have found the DiskMask method to be fitting for the reasons stated in Section 1 [3]. We have already established a general understanding of how DiskMask works in Section 2.3. This chapter focuses on the implementation of DiskMask in PyTorch and gives insight about the underlying concepts and how they were transferred into code. In addition, details about the difference in the coding strategy between MatCaffe and PyTorch are given and broken down for the reader, creating an in depth understanding of the DiskMask method as well for possible future performance improvements and implementations in different programming languages.

3.1 Data Preparation

The most challenging part of the PyTorch implementation was to prepare the data and the weight map, reference point and disks. It was really important to get the same outputs from the *prepareData_f.m* function with a reasonable performance since the training depends on the accuracy of these outputs.

3.1.1 Data Augmentation

Our first approach was to replicate the same tactic as MatCaffe implementation for the data augmentation. In MatCaffe, *augmentData_v2_f.m* class uses a grid and all of the augmentations such as random rotation, shift, and flipping which are the augmentations used on C. Elegans are applied on this grid. This grid is just an intermediate construct and it is then applied to the image by an *interpn* [10] operation to interpolate the input image at the query points defined by this grid. This operation results in the augmented image. Our approach in *datasetUtiliy.py* was to create a transformation class, *AugmentData_v2_f*, to be given as an input to the PyTorch dataloader to be applied on images when iterating through the dataloader. The reason for creating this transformation class instead of a simple Python method was to take advantage of the batch size parameter so that one could train in batches if

they wanted to even though DiskMask was trained with batch size of 1. We made use of the NumPy [11] and SciPy [10] classes to replicate the behaviour of the operations on the grid. Although this transformation was working successfully, we had to replace it in the end because of performance issues. It takes 1.2 seconds to apply a set of augmentations to a single image. We found out this was because of the Python version of the *interpn* operation. *RectBivariateSpline* [10] was used to get the same results as the Matlab *interpn* function. This operation proved to be very slow when applied channel-wise to the ground truth segmentation masks, especially when the degrees of the bivariate spline was set to 3 in order to perform bicubic interpolation. The fastest method to replicate the *interpn* method was to use the *map_coordinates* [10] method from Scipy for all of the interpolations and not just the ones using nearest neighbor interpolation, exchanging accuracy for speed, but it resulted in very inaccurate transformations. We still present this transformation class for possible future performance increase but we also needed to find another way of doing the data augmentations.

Another way to do the data augmentations is to make use of the *transformations* class of *Torchvision* [6] and apply the regular transformations such as random flipping, affine transforms (Rotation and shifting), center crop in sequence to get the same results as the Matlab implementation. The transforms can be found in the *CombinedTransforms* class with the options to pad and mirror the borders as in the Matlab implementation. This way the augmentation take 0.3 seconds and it is much more feasible to train with high number of epochs. These augmentations are also applied to the ground truth segmentation maps and the weight maps with identical parameters as in the Matlab implementation

3.1.2 Custom Dataset Class

Instead of reading the data in the training script as in the Matlab implementation, we make use of a custom dataset by subclassing the Pytorch *torch.utils.data.Dataset*[6]. This way the custom dataset class *EleganSDataset* can be called once before training and take the Pytorch transformations mentioned in Section 3.1.1 as input. This allows for the option of using mini-batches with arbitrary size and utilizing the GPU by using many workers very easily. Although we use a batch size of 1 here, this feature adds convenience for future works. Batches of images, labels, classes and weights are read from files and prepared in the `__init__` function of the custom dataset as `batchIms_c`, `batchLabs_c`, `batchCls_c` and `batchWghts_c`. The sampling

distribution function *precomputeSFD_f* is also computed only once before the training in the `__init__` function of this custom dataset class.

3.1.3 Preparation Of Weight Map, Reference Points and Disks

From the augmented data, variables such as the weight map, reference points and disks that are required for training are prepared in the *prepareData_f.py* class. This class uses many methods which were implemented by using the Numpy and Scipy counterpart functions for finding and ignoring the instances on the borders, setting the reference points and computing their size, weighting the space between the disks, cropping the ground truth segmentation masks and the weight map to the correct tile size. Additionally, OpenCV [12] and Skimage [13] were used to replicate operations such as *imerode* and *bwlabel*.

3.1.4 Helper Functions

Helper functions in the context of data preparation are the functions that are used to prepare the results that take the augmented images as input and generate the inputs that are required for training. We will not go into detail about every single piece of Python implementation but only for some processes that might not be very obvious from the first glance.

We are skipping the implementation details of *ignoreMasksOnBorders_f* since it is composed of simple matrix operations and library operations with the same names between Matlab and Python. The ground truth segmentation masks and an array of the classes of these segmentation masks (All 1's in this case because we are doing instance segmentation on the same class of objects, the C. Elegans worms) are input into *ignoreMasksOnBorders_f* and a list of the ground truth segmentation images that do not touch the input tile borders is outputted.

setRP

The reference points are placed relative to each of the ground truth segmentation mask in this function as explained in Section 2.3. As in the Caffe implementation, we use a variable `rp_id` to choose where we want to set the reference point at. We use 0 as input in this case which means we are setting them at the centroid, the center of mass, of the objects. We take the ground truth segmentation maps `masksRPs_c` as input and output a list with the reference points. The individual reference points are then projected into a single channel dimension using

```
mRPs = np.argmax(np.concatenate(((np.ones(mRPs[:, :, 0].shape, dtype=
                                  int) * 0.1)[:, :, np.newaxis], mRPs
                                  ), axis=2), axis=2)
```

where `mRPs` are the reference points on the grid. The generated reference points are used to generate the disks in *genDsksOfadaptSize_f*.

genDsksOfadaptSize

There are three main parts of this function that might not be obvious from the code.

First difference is the use of both *skimage.footprints.ellipse* and *skimage.morphology.disk*[13] to emulate *strel('disk')* from Matlab. This is because both of the skimage functions handle disks with radiuses 1 and 2 differently. When we use *strel('disk')* for disk radiuses 1 and 2 and *skimage.morphology.disk* for disk radiuses larger than 2, we can emulate the behaviour of *strel('disk')* identically. Since the disks have adaptive sizes in DiskMask, when two disks are close and they are shrunk by the algorithm, it is useful to have the exact same representation of disks because every pixel gets a different entry in the weight map and every pixel difference is valuable for overlapping objects with reference points that are very close to each other. The difference in generated disks for different sizes of radiuses can be seen in 4.

Second difference is the `if (len(m_RPD[tmp_diffMask ==1])) > 0` addition. This check before calculating the new disk radius. This replaces the `if isempty(disk_radius_n)` check from the Matlab code since the Matlab disk radius calculation returns an empty array while the Python version gives out an error. In the end both of the operations are identical in practice.

Third difference is the use of *scipy.ndimage.morphology.distance_transform_edt* in Python to replicate the use of *bwdist* in Matlab. We needed to input `1 - np.where(mRPs +1 > 0, 1, 0), return_indices=True` to the Python version as opposed to just a simple `mRPs` as in the Matlab version because *distance_transform_edt* uses 0 elements for the background so we need to flip the 0's and 1's. In short, we can attain the same euclidean distance transform in Python with this change.

As a result, *genDsksOfadaptSize_f* outputs three outputs which are `dskswCl`, `rpIDs` and `rpWgtsInsDsks`. These are the generated disks, reference points with ID's for different object, and the weighted disks according to their sizes to be used in the weight map respectively. The outputs are represented in Figure 5.

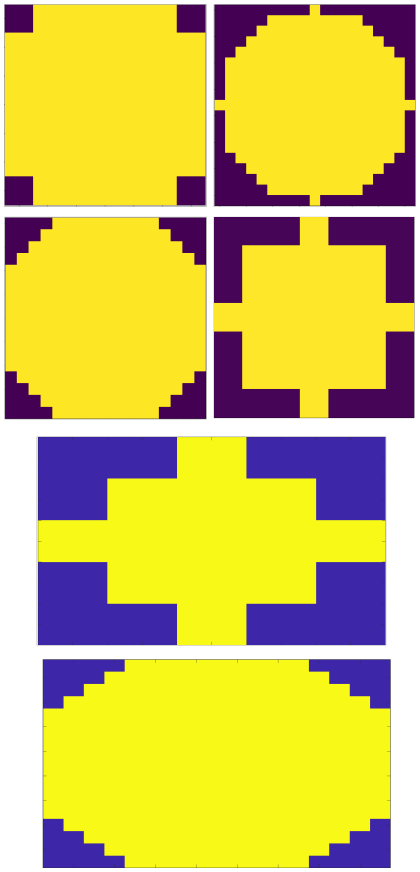
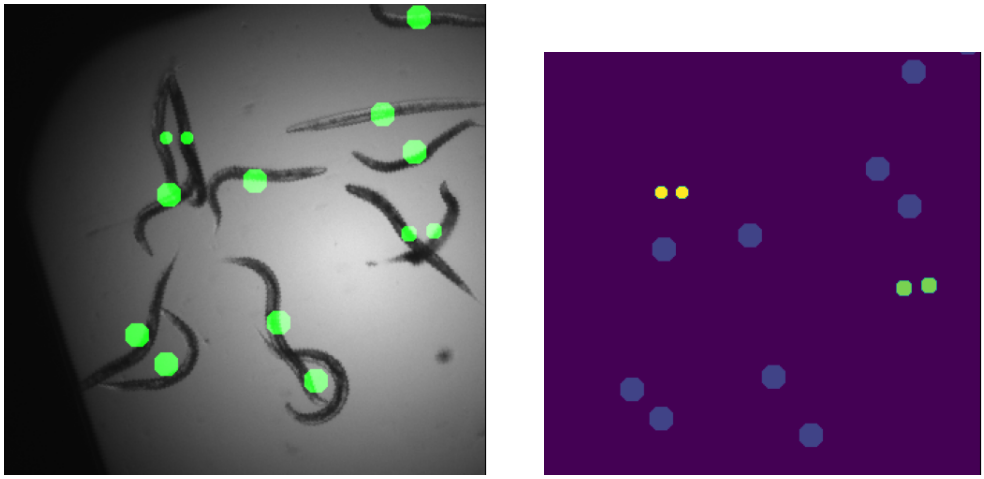


Figure 4: Generated disks with different radiuses and different functions for Matlab and Python. The first two outputs are the incorrectly generated disks for small and large radiuses respectively. The following two disks are the correctly generated disks that provide identical output to Matlab disks which are the last two images.

cropBlob

Although blob is a Caffe [5] concept, we kept the same name for establishing correspondence between the Matlab and Python version easier. The same method that was used in Section 3.1.1 for doing transformations on a grid is also used in this part, namely *applyOpChannelWise*, which gets *myImcrop* as input to apply the crop operation to all the channels of ground truth segmentation maps. The rest of the Python implementation is straightforward and uses the Numpy equivalent functions.



(a) C. Elegans With Generated Disks **(b)** Disks With Assigned Weights

Figure 5: Generated disks. In (a), the disks that are close to each other have their size adapted to be smaller so that they do not intersect. In (b), the brighter yellow color means higher values of weight. Disks that are far from each other have weights of 1, the closer ones (Green) have 4, and the closest ones (Yellow), have 5. In short, disk size and weight of disks are inversely proportional.

weightSpaceBtwDsks

This function is used to generate the weights in the space between the disks as described in DiskMask to "...ensure their spatial isolation"[3]. The Python implementation uses basic Numpy operations to replicate behaviour identical to Matlab implementation. The *skimage.measure.label*[13] is used to assign label number to disks. This is identical to the *bwlabel* operation in Matlab. We need to input `dskswCl, connectivity=2` to the function to use a 2 connectivity as it is the default behaviour for Matlab *bwlabel*. An example of the generated weights between the disks can be seen in Figure 6.

precomputeSFD

This function is used to calculate the sampling distribution as in the Matlab implementation of DiskMask. The ground truth segmentation masks are projected to a single channel to form the semantic segmentation masks and are passed through a Gaussian smoothing kernel to sample tiles with more object density in order to accelerate the training. An example of the ground truth segmentation maps can be seen in 7.

The main difference between the Pytorch and Matlab implementations is the

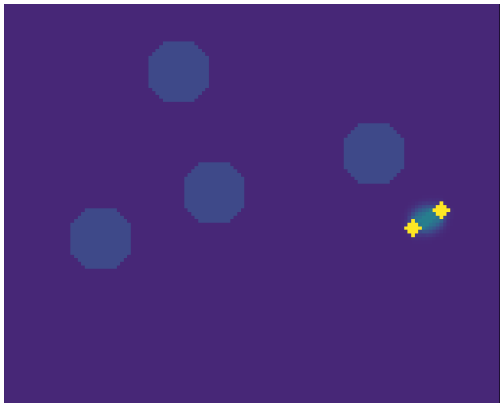


Figure 6: Weights between disks. The weights of the space between the disks for the disks that are very close to each other is increased to force the network into learning the small space between the disks as suggested in the U-Net paper[4]. The yellow disks are very close to each other so the weight of the background between them is increased in this weight map.

2-D Gaussian filtering function *imgaussfilt*. There are two ways to replicate the *imgaussfilt* function behaviour from Matlab. Firt is to use the OpenCV approach `cv2.GaussianBlur(tmp_sfD, ksize=(0, 0), sigmaX=50, borderType=cv2.BORDER_REPLICATE)` where `tmp_sfD` is the ground truth segmentation masks collected into a single channel. This approach gives us a result very close to Matlab but to replicate the function identically we use *skimage.filters.gaussian* with `skimage.filters.gaussian(tmp_sfD, sigma=50,mode ='nearest',truncate=2.0)`. A representation of the smoothing results can be seen in 8.

3.2 The Model

3.2.1 Defining The Model

Models are defined in a Prototxt file in Caffe. In Pytorch, however, they are defined as a class thats subclassing the *torch.nn.Module* [6] class. We create the class `DiskMaskNet(nn.Module)` for defining the model in the PyTroch implementation.

We implement each convolution block in 3 separately and connect them together in the `forward()` function instead of creating a big *torch.nn.Sequential* module since we want to make use of the skip connections as in figure 3. As an example, the PyTorch implementation of the first convolution block in the EncoNet part is given in the code segment below.

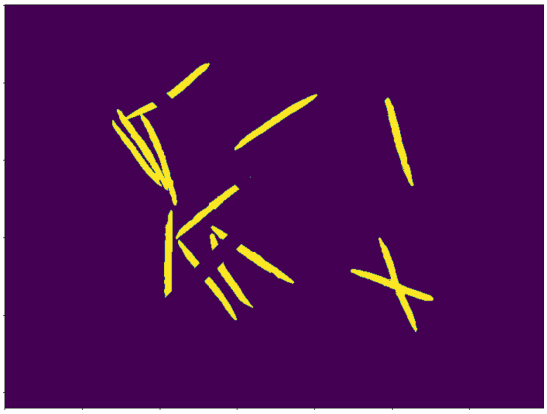


Figure 7: Semantic segmentation mask. The ground truth labels of each object projected into a single dimension to create the semantic segmentation mask

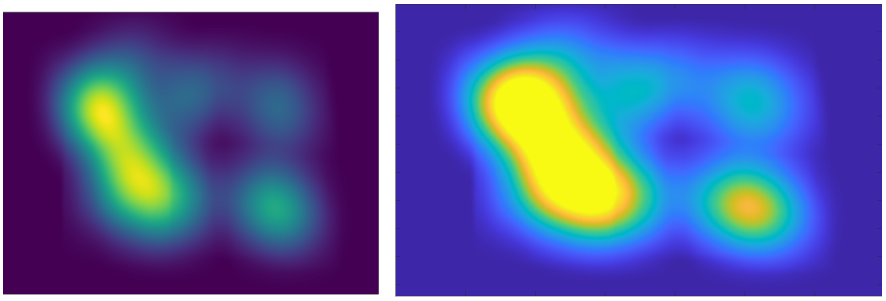


Figure 8: Segmentation mask through Gaussian filter. The sampled tiles during training are chosen from the sampling distribution acquired through passing the segmentation mask through a Gaussian filter. The image on the left is from *skimage.filters.gaussian* in Python and the image on the right is from *imgaussfilt* in Matlab. Both outputs are identical by value.

```
self.n1_d1c = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=64, kernel_size=(5, 5),
                                        stride=2, padding=0),
    nn.ReLU(),

    nn.Conv2d(in_channels=64, out_channels=64, kernel_size=(3, 3),
                                        padding=0),
    nn.ReLU(),
    nn.Conv2d(in_channels=64, out_channels=64, kernel_size=(3, 3),
                                        padding=0),
    nn.ReLU(),
)
```

Which is composed of unpadded convolution and ReLU operations in a sequence. As an example, the last layer of the same convolution block defined in Caffe Prototxt file which is also composed of a convolution and ReLU layers following each other is given in the code segment below.

```
layer { bottom: 'n1_d1b'  top: 'n1_d1c'  name: 'n1_conv_d1b-c'  type: 'Convolution'  param { lr_mult: 1 decay_mult: 1 } param { lr_mult: 2 decay_mult: 0 } convolution_param {  num_output: 64  pad: 0  kernel_size: 3 kernel_size: 3  weight_filler { type: 'msra' }} }
layer { bottom: 'n1_d1c'  top: 'n1_d1c'  name: 'n1_relu_d1c'  type: 'ReLU' }
```

With the same approach, we replicate the whole model in `DiskMaskNet`. The only detail to be mindful of is that the crop and concatenate operations (Skip connections) between and inside the encoder-decoder networks can be included in the Caffe Prototxt file along with the model definition while they have to be handled separately in the `forward()` function in PyTorch, described in 3.2.2.

3.2.2 Forward Pass

We implement the forward pass in the `forward()` function by connecting the layers we defined in section Section 3.2. We pass the signal through the layers by passing the outputs of a convolution block to the following one as can be seen in the code snippet below.

```
#Encoder Part
n1_d1c = self.n1_d1c(x)
n1_d2c = self.n1_d2c(n1_d1c)
n1_d3c = self.n1_d3c(n1_d2c)
n1_d4c = self.n1_d4c(n1_d3c)

#Decoder Part
n1_u3d = self.n1_u3d(n1_d4c)
n1_u2d = self.n1_u2d(n1_u3d)
n1_u1d = self.n1_u1d(n1_u2d)
```

We also handle the cropping and concatenation operations required for the intra-net and inter-net skip connections. The intra-net connections can be established by the example code snippet below.

```
n2_u1a = self.n2_u1a(n2_u2d)
n2_d1c_cropped = n2_d1c[:, :, 40:40 + n2_u1a.shape[2], 40:40 +
                                    n2_u1a.shape[3]]
n2_u1b = torch.cat((n2_u1a, n2_d1c_cropped), dim=1)
```

The inter-net skip connections can be implemented by saving the outputs from EncoNet and concatenating them when the forward pass signal is passing through the respective DecoNet convolution and deconvolution layers as in the example code snippet below. The `n2_d1c` is the corresponding convolution layer from EncoNet.

```
n2_u1a = self.n2_u1a(n2_u2d)
n2_d1c_cropped = n2_d1c[:, :, 40:40 + n2_u1a.shape[2], 40:40 +
                                    n2_u1a.shape[3]]
n2_u1b = torch.cat((n2_u1a, n2_d1c_cropped), dim=1)
```

The inputs of the forward function are `x` and `gates` where `x` is the input image and `gates` is the activated gateways that is explained in Section 2.3.1 where there is a 50% chance of opening any gateway for an object during training while only a single gateway is opened during inference time.

3.2.3 Weight and Bias Initialization

We can see from the Caffe Prototxt file that the original DiskMask implementation uses *msra*[5] weight initialization for every convolution and deconvolution layer. This initialization uses the method from [14], filling in the weight tensors according to a normal distribution $X \sim N(0, \sigma^2)$ where σ is the standard distribution

$$\sigma = \sqrt{\frac{2}{n}}$$

In the Caffe description, n is the fan_in, fan_out or their average depending on the chosen mode where fan_in is the number of inputs to a layer and fan_out is the number of outputs going out from the layer. The default mode in fan_in so the n is the number of inputs coming into the layer. The identical initialization that also uses [14] in PyTorch is *torch.nn.init.kaiming_normal_*. We initialize the weights of every convolution and deconvolution layer by creating a method to parse through each layer in the model and checking if the layer is a convolution or a deconvolution layer to

initialize their weights with Kaiming method. We also zero out the biases of every convolution or deconvolution layer within this method. We use the built-in `apply` method of the model to get this method as an input. The method can be seen below.

```
#MSRA weight initialization in caffe file
def weights_init(m):
    if isinstance(m, nn.Conv2d) or isinstance(m, nn.ConvTranspose2d):
        nn.init.kaiming_normal_(m.weight.data)
        nn.init.zeros_(m.bias.data)
```

The weights are applied to the model by a simple `model.apply(weights_init)` operation where `model` is an object of class `DiskMaskNet(nn.Module)`.

3.3 Training

The training of the model takes place in the `main()` function of the *diskmaskport_single_gpu* class.

3.3.1 Setting Up The Environment

In the Caffe implementation, the loss functions are defined in the Prototxt file along with the model definition and the optimizer can be found in a seperate *solver* Prototxt file. In the PyTorch implementation we define these before the training loop takes place.

Model

We start by defining the model by creating an instance of the `DiskMaskNet` class. Then we put this model into the `device` which is the GPU of the device we are training on. We can do this by a simple line of code `model = DiskMaskNet().to(device)`.We then initialize the weights as described in Section 3.2.3.

Optimizers

We then define our optimizer as described in the original DiskMask paper, with the ADAM[15] optimizer and starting with the learning rate `lr = 0.0001` and `betas = (0.9, 0.999)`. We also define the learning rate scheduler that takes our optimizer object as input in order to reduce the learning rate from 0.0001 to 0.00001, a multiplicative factor of 0.1, after the 300000'th step as described. We define the

multiplicative factor in the `gamma` parameter of the scheduler. The full optimizer and scheduler definitions can be seen below.

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001, betas=
                                    (0.9, 0.999))
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=
                                    300000, gamma=0.1)
```

Loss Functions

The next thing we need to define is the loss functions. EncoNet's output is trained with the *SoftmaxWithLoss* layer in Caffe, which is just a softmax layer followed by a logistic loss layer. We can get the identical results in PyTorch by using the *torch.nn.CrossEntropyLoss* which computes the log softmax internally and passes these results through a logistic loss. Since we need to ignore the pixels in the `ignore` map as described in the Caffe Prototxt file, we also use the `ignore_index=255` parameter so that the pixels with values of 255 are not computed into the loss and they will not affect the training. We also use the `reduction = 'none'` parameter so the losses are given in a grid instead of a single value of the mean of this grid. We do this so that we can use the weight map we have computed from the ground truth segmentation masks as explained in Section 2.2.3 and then reduce the loss to get a single value.

For DecoNet's loss function, *SigmoidCrossEntropyLoss* is used in the Caffe implementation, which is a combination of a sigmoid layer and a cross-entropy layer. In PyTorch, we use *torch.nn.BCEWithLogitsLoss* which does the same combination to give a numerically more stable computation. This loss function does not have an `ignore_layer` argument as in the loss function of EncoNet, so we have to use the `reduction = 'none'` in order to get the loss map for every pixel and then manually zero out the losses with corresponding `ignore_label` values which are 255 in our case before taking the mean of the loss map. In short, the reasoning for the lack of reduction is different for both of the loss functions, it is to use the weight map in EncoNet and to manually implement `ignore_label` logic in DecoNet. The definitions of the loss functions are as below.

```
enco_criterion = nn.CrossEntropyLoss(ignore_index=255, reduction='
                                    none')
deco_criterion = nn.BCEWithLogitsLoss(reduction='none')
```

Training and Transformation Options

Next, we define the `opts` variable that controls all of the training and transformation options such as `disk_radius` or `deform_magnitude` in a dictionary so that we can easily change them.

Dataset and Augmentations

As mentioned in Section 3.1, we offer two different implementation strategies for the data augmentation part. First one is the one that is identical to the Caffe implementation where the transformations are applied on a grid in sequence to be finally applied to the input image in the end as discussed in Section 3.1.1. This is implemented in PyTorch in *AugmentData_v2_f* class. We decided to not use this implementation strategy because of the running speed of the `RectBivariateSpline` operation. The second strategy is to use the PyTorch transformation operations in sequence to get the same transformations without using a grid, which is done in the *CombinedTransforms* class. We use the latter strategy because of the much faster running speed. We create an instance of the dataset as below.

```
elegans_dataset = ElegansDataset(data_dir=data_dir, transform=
                                 CombinedTransforms())
```

3.3.2 Training Loop

In the training loop we have a total of 600000 epochs as described in the original DiskMask paper. We stay true to the original number of epochs so that we can compare the evaluation performance of the trained PyTorch model to the Caffe model with the same amount of training.

Running Loss

First off, we keep track of the `running_loss` which is the total loss over the augmented images. If the number of augmented images per image is 1 then this variable is equal to the loss of a single backward pass. We can change the number of augmented images generated per input image in the `augment_opts` dictionary.

Loading Data

Loading the data takes place entirely in the `loadData` function by using the inputs `elegans_dataset` and `augment_opts` which are discussed in detail in Section 3.1.

The outputs of this method are the augmented images, augmented ground truth segmentation masks and augmented weight maps which go through identical transformations. The method calls `prepareData_f` from Section 3.1 within itself to get the augmented data and weight map, it is just used to connect the dataloader with the dataset. The method is called as below for the C. Elegans.

```
worker_id, augData_c, augLabs_c, augWeights_c = loadData(0,
                                        elegans_dataset, augment_opts)
```

The first argument 0 is the `worker_id` and it is just a placeholder and is required for the parallelization of the data loading operation. We do not use parallelization in the PyTorch implementation since we decided to use the PyTorch transforms with `CombinedTransforms` class instead of the grid transforms with `AugmentData_v2_f` as explained in Section 3.1. We can have parallelization in our dataloader by having the `num_workers` increased and using a higher `batch_size` to perform the transformations in batch with multiple workers instead.

We then iterate through all of the augmented data generated from a single image and the rest of the implementation takes place in this loop. We first get the single image from the augmented images and replicate the channel dimension into 3 channels if the image is grayscale and only has 1 channel with the code segment below.

```
im = augData_c[r_d]
if im.shape[2] == 1:
    im = np.tile(im, (1, 1, 3))
```

Preparation Of The Model Inputs

To begin, we create the ignore map by `ignore = wghtCls == 0` as in the Caffe implementation of DiskMask. Then, we handle the segmentation in ignore regions with array slicing operations. C. Elegans does not have object in the ignore regions but we still offer the implementation to be used on other datasets again by using array slicing operations. We assign the ignore labels to `dskswCl` by using `dskswCl[ignore ] = 255` so that we can ignore these pixels that are essentially pixels that are out of the image borders resulting from image augmentations. We can also create custom ignore masks if we wanted to by manipulating the `ignore` array. The `label_2D_cls` is named after the Caffe implementation and it is one of the bottom blobs to calculate the EncoNet loss in Caffe as well as to input into our `enco_loss` in the PyTorch

implementation. It is converted into a PyTorch tensor and assigned to the GPU device.

```
    label_2D_cls = torch.tensor(dskswCl).to(device, dtype=torch.long)
```

We label the disks with the same method as in Section 3.1.4 to later assign them to different channels and create the segmentation map and assign these to the `ccDsks` variable. We then locate the reference points in the borders, we would like to exclude their disks when we are training since their centroid might be outside of the input image tile. This is done by the `detectRPsOnBorders_f` function which takes the reference points with ID's corresponding to the respective segmentation mask channel `rpIDs`, and the disks belonging to these reference points `ccDsks`. We will not give many implementation details about this function since it is composed of basic NumPy operations.

```
    rp_r2k_m, ref2reject = detectRPsOnBorders_f(rpIDs, ccDsks)
```

The next function we use is the `activateRPandMask_v2_f` which activates a subset of the gateways for the training as decribed in Section 2.3.3. The activation probability can be changed with the `augment_opts['keep_prob']` argument but for the C. Elegans training we use 0.5 chance to activate each gateway. The gateways of the disks on the borders are excluded. We also will not give the implementation details for this function since it is composed of basic NumPy operations. We use the variables we have prepared up to this point from Section 3.3.2 as inputs to this function to get the segmentation maps `augLabs` and disks `bwDskswCl` belonging to the activated gateways.

```
augLabs, bwDskswCl = activateRPandMask_v2_f(rpIDs, augLabs, ccDsks,
                                    bwDskswCl, ref2reject, augment_opts
                                    ['keep_prob'])
```

The disks belonging to the activated gateways are taken and they go through a random thinning operation to reduce their size. This is done to prevent the network from over-fitting to a certain size as explained in Section 2.3.

```
bwDskswCl = thin(bwDskswCl, random.randint(1, augment_opts['
                                    disk_radius'] - 1) - 1)
```

At last, we convert the inputs we have prepared into tensors for GPU training and we arrange their channels accordingly for the network to accept them.

```
im = torch.unsqueeze(torch.permute(im, (2, 0, 1)), 0).to(device, dtype
                                        =torch.float)
bwDskswCl = torch.tensor(bwDskswCl).to(device)
```

Training The Model

In this subsection, we describe the PyTorch implementation details about the training process such as the loss functions, scheduler, optimizer and loss. We input our image tile `im` and the activated gateways `bwDskswCl` from the previous subsection into our model.

```
pred_cls, pred_semseg = model(im, bwDskswCl)
```

The outputs `pred_cls` and `pred_semseg` carry the same names as the blobs in the Caffe Prototxt file. These are the only two outputs of the image. The `pred_cls` output is one of the two outputs of EncoNet, the other being the feature map F that is an input of DecoNet, and it is the predicted disk logits with 2 channels which is determined by the number of object classes (C=2 in this case, the objects and the background) as described in DiskMask[3]. The second output `pred_semseg` comes from the DecoNet part and it is the predicted segmentation mask.

To train the DecoNet part, we need to format the ground truth segmentation masks. We use the `cropBlob_f` function from Section 3.1.4 to crop the ignore map and ground truth segmentation masks to the DecoNet output size. We use the ignore map to change the ground truth segmentation mask's ignore regions to 255 with a slicing operation as below.

```
augLabs[ignore.astype(int) == 1] = 255
```

This is the final state of the ground truth segmentation masks to be input into the DecoNet loss function. We convert this array to a tensor and assign it to the GPU as `gt_semseg`.

In the Caffe implementation of DiskMask, the loss functions are handled in the Prototxt file, here we need to handle them in the training loop. We input the `pred_cls` and `label_2D_cls` that we have prepared before as inputs to `enco_criterion` that we defined in Section 3.3.1. Since we sent the reduction operation to *None*, we take the resulting loss map and multiply it element-wise with `wghtCls` which is the weight map described in Section 3.1.3 and the U-Net paper[4]. Since it is unreduced data, we reduce it with a mean operation as below.

```
enco_loss = (enco_loss * torch.tensor(wghtCls).to(device).unsqueeze(0)
                                    ).mean()
```

For training the DecoNet, we use the `deco_criterion` that was defined in Section 3.3.1. We input `pred_semseg` and `gt_semseg` that were defined in this subsection earlier which are the predicted and ground truth segmentation masks respectively. Since *torch.nn.BCEWithLogitsLoss* does not have an ignore label implementation, we implement it ourselves by taking the unreduced loss map `deco_criterion` and assigning the elements which are 255 in `gt_semseg`. This way we ignore the losses that were calculated from ground truth masks that were supposed to be ignored as below.

```
deco_loss[gt_semseg.unsqueeze(0).unsqueeze(0) == 255] = 0
```

We then reduce the losses from the resulting DecoNet loss map by summing up all of the losses and dividing them by the number of pixels in the segmentation masks that are not 255. In short, we are taking a mean of the losses of pixels that are not ignored as below.

```
deco_loss = deco_loss.sum() / (gt_semseg.unsqueeze(0).unsqueeze(0) !=
                                    255).sum()
```

Then we sum up these resulting losses since both the EncoNet and DecoNet losses have a weight of 1 in the Caffe Prototxt file and they are weighted equally. This becomes our total loss that we train on. Lastly, we do a backward pass on this total loss with the optimizer step. We also do a scheduler step so that we can reduce the learning rate by 10% after 300000'th epoch. We also save the model every n'th epoch of our choice.

3.4 Inference

For the inference part, we implement the *diskmask_ inference* script which implements the *tilePredictDtc* and *tilePredictSgm* functions as in the Matlab implementation.

3.4.1 Main Inference Script

This script is a connection point for the two other inference methods which are *tilePredictDtc* and *tilePredictSgm*. We begin by loading our test data from the dataset folders and putting the data into formats that are accepted by the methods that

(a) Predicted Soft Segmentation Mask

(b) Predicted Segmentation Mask

Figure 9: Soft To Filtered Scores. In (a), we have the segmentation mask with soft scores. In (b), we have the segmentation mask filtered from the soft scores by taking the pixels with scores over 0.5.

follow. We load our data into `images_c` which are the input images. We do not need the ground truth segemntation masks, weight maps or class labels since we do not use any loss functions. At the rest of this section, we iterate through the input images and the explained operations are all done on single data point.

First, we use the `tilePredictDtc_f` to pass the image signal through the network and predict the object disks. We pass the single image `im` and the options `test_opts` as input arguments to this function to get `scores`. We use the `medfilt2d` function from *scipy.signal*, which has a function with the same name in Matlab, to do median filtering on the normalized `scores`. We then label each disk as in the previous sections. Then, the concatenation of the input image and the labeled object disks are passed as input to `tilePredictSgm_v2_f` to output the new `scores` to predict the segmentation masks.

Lastly, we take the output `scores` to calculate the objectness/confidence scores. The objectness scores are calculated by taking the soft segmentation masks and thresholding them by 0.5 on the `score` variable. The mean of the elements of the soft prediction masks that have value over 0.5 is used to calculate the objectness/-confidence score. This score is useful for calculating the average precision (AP) in Section 4. The predicted segmentation mask is acquired by creating a new array where `masks_soft_pred >=0.5)` can be seen in 9. In the end, the outputs are saved into a hdf5 file.

3.4.2 Predicting Reference Points and Object Disks

The `tilePredictDtc` function is used for predicting the object disks through EncoNet. We initialize the model with the same settings as we did in Section 3.3 but with the exception of loading the state dictionary from the trained model's pt file instead of initializing the layer weights with the `kaiming_normal` weights. For getting the outputs from the model, Caffe uses the Matcaffe interface to get the value from a layer as blobs by the code segment below.

```
solver.test_nets(1).forward({paddedInputSlice});
```

We need to use the `DiskMaskNet` instance that we created to do the same thing. Since the model we created needs to have the gateways, we pass a `gates` variable with placeholder values of 1's. The EncoNet output does not actually need the `gates` to be computed so these are just mock values for the model to not have missing arguments. This can be seen in the code segment below.

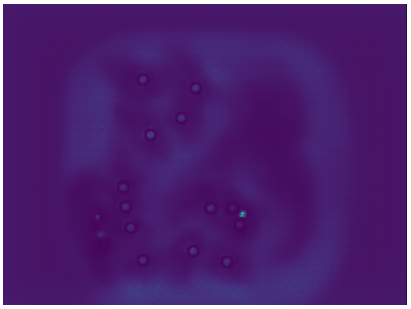
```
mock_gates = torch.ones((324, 324)).to(device)
pred_cls, pred_semseg = model(torch.permute(torch.tensor(
                                    paddedInputSlice).unsqueeze(0), (0,
                                     3, 1, 2)).to(device), mock_gates)
```

The rest of the function is the same as the Matlab implementation but with Numpy array operations. Although we do not use it for the C. Elegans experiment, we also offer the mirror padding options like in the Matlab implementation. The resulting disk logits have 2 channels (Foreground and background classes) and we take the index of the maximum element of these logits across the channels to decide which class the pixel belongs to as in the Matlab implementation. An example is given in 10 for the following operation.

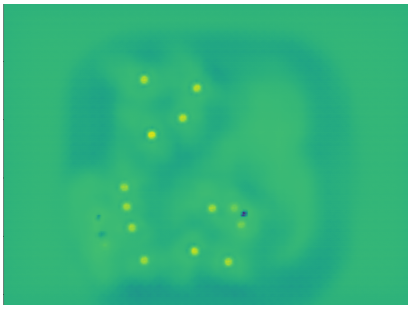
```
pred_cls = np.argmax(np.squeeze(scores[0]), axis=2)
```

3.4.3 Predicting Segmentation Masks

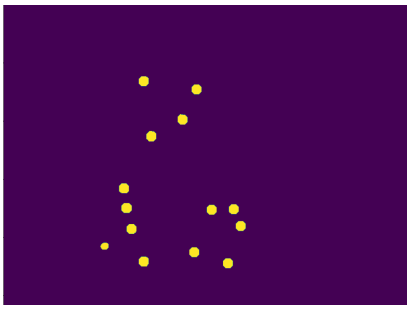
The `tilePredictSgm_v2_f` function is used for predicting the segmentation masks through DecoNet. As in Section 3.4.2, we implement the same operations as the Caffe implementation using Numpy operations. This time, we use the mock gates as in Section 3.4.2 to get the output from the EncoNet, but we let the data pass through the network a second time by calling the model again with these parameters from the first time. The predicted disks from the first pass are used to predict the segmentation



(a) Disk Logits From First Channel



(b) Disk Logits From Second Channel



(c) Disk Masks

Figure 10: Disk Logits To Disk Masks. In (a) and (b), disk logits from the first and second channel are shown respectively. In (c), disk logits generated from the `argmax` operation are shown.

masks. In the Caffe implementation, this is done by creating two networks, one for EncoNet and one for DecoNet, to transfer values between them using blobs as below.

```
solver.test_nets(2).blobs(opts.expectedPredData{c_i}).set_data(solver.
                                    test_nets(1).blobs(opts.
                                    expectedPredData{c_i}).get_data());
```

In the Pytorch implementation, we save the results from the first pass and pass through the network a second time for the segmentation masks. As in the DiskMask paper[3], we activate only one gateway during inference time and input this gateway as the `gates` variable in the second pass. The code segment below shows how we do the second pass with `m_bin` being a single one of the predicted disks from the first pass.

```
m_bin = pred_cls == un_i
pred_cls2, pred_semseg2, expected_pred_data2 = model(torch.permute(
                                    torch.tensor(paddedInputSlice[:, :,
                                     :3]).unsqueeze(0), (0, 3, 1, 2)).
```

```
                                    to(device), torch.from_numpy(np.
                                    transpose(m_bin, (3, 2, 0, 1))).to(
                                    device))
```

3.4.4 Average Precision

We compute the average precision according to the COCO[16] average precision calculation rules as described in [17]. We implement a script *evaluate_results* to do these calculations. The ground truth segmentation masks and predicted segmentation masks are taken from Section 3.4. We load the labels in a sorted way to match the predicted masks and loop through all the predicted masks for a single image. We take the Jaccard index, also known as intersection over union (IOU), between the predicted masks and ground truth masks. The IOU score is checked against the thresholds from 0.5 to 0.95 as described in the DiskMask paper and the true positive and false positive results are computed from this check as below.

```
iou_score = jaccard(torch.from_numpy(pred_conf[1]), torch.from_numpy(
                                    pred_conf[0]))
if iou_score >= 0.5:
    tp += 1
else:
    fp += 1
```

Precision and recall metrics are calculated from the true positives, false positives, and total number of ground truth masks in the image. These scores are saved in a tuple along with the objectness/confidence scores from Section 3.4.1 and they are sorted by the confidence scores as decribed in [17]. We then draw the Precision-Recall curve with the same order and take the area under the curve to get the average precision. Lastly, we average these average precision values over the classes to get the singular mean average precision, which is used interchangibly with average precision as in the COCO evaluation[16]. We report the resulting values in Section 4.

4 Experiments

For the experiments section, we replicate the same setting with DiskMask[3] on the C. Elegans[2] dataset.

4.1 Data

As in the DiskMask experiment, C. Elegans[2] live/dead assay with 100 images separated into 25 test, 25 validation and 50 train images. The 16 bit data is projected into 2-D and replicated for 3 channels (RGB). Data normalization is done as in the original DiskMask paper by dividing the images by 3200 to get the pixel values to range [0, 1]. Rotation, flipping and cropping are applied before training. Ground truth labels are separated into different folders for different images for easier evaluation and training.

4.2 Metrics

We use average precision to evaluate the model. The process is described in detail along with the implementation details in Section 3.4.4. Thresholds from 0.5 to 0.95 are used as in the original DiskMask paper[3].

4.3 Hyperparameters

For the training, we use the ADAM optimizer with beta values $(0.9, 0.999)$ and a learning rate of 0.0001. We use a scheduler to reduce the learning rate by 10% at the end of the 300000'th epoch as in the DiskMask paper. We train the network for a total of 600000 with a batch size of 1.

4.4 Result

Results are presented both quantitatively and qualitatively.

4.4.1 Qualitative Analysis

An example of inference results on the test dataset can be seen in 11. The predicted segmentation maps are colored for better visibility.

4.4.2 Quantitative Analysis

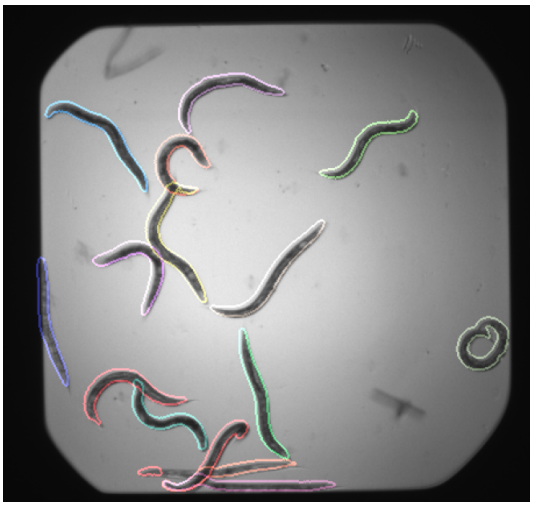
The average precision values are reported in 1 along with the original DiskMask results on the same experiment.

	AP	AP_{50}	AP_{75}	AP_{90}
Caffe Implementation[3]	.804	.947	.916	.592
PyTorch Implementation	.778	.926	.921	.469

Table 1: Average Precision Values. The first column contains the average of AP values from 0.5 to 0.95. The following columns contain AP values with thresholds 0.5, 0.75 and 0.9 respectively.

Although the AP results are generally very close to the original Caffe implementation results, we can see that the model starts to fall behind a considerable amount when we start to increase the threshold to close to 1. From this, we can infer that the PyTorch model performs quite good in general but falls behind in more detailed segmentations. This might be due to the difference between the implementations of the training loops and internal implementation differences of Caffe and PyTorch methods. Another possible reason is the difference in the image augmentation strategy.

(a) Image With Predicted Masks



(b) Image With Predicted Masks2

Figure 11: Images with predicted masks. In (a), we see that the segmentation masks are well seperated even when there are overlaps in the objects. There are still shortcomings of the model, the object in the bottom left corner did not get detected. In (b), we have another image with good overall segmentations apart from the triple masks in the bottom. The network seems to be not performing good enough when there is too much clutter from multiple objects.

5 Conclusion

The ineffectiveness of the bounding-box method against elongated objects with non-geometrical shapes requires other types of solutions to the popular instance segmentation problem. DiskMask[3] proposes a new method that uses disks at the centroids of the objects instead of objects. Their method proves to be highly effective with the instance segmentation of elongated and overlapping objects. In this work we investigate instance segmentation of overlapping objects and offer a new PyTorch implementation to DiskMask[3]. We implement a convenient PyTorch version to enable future research that does not use bounding-boxes. We aim to have performance close to the original implementation so that future research can utilize the advantages of PyTorch without sacrificing performance. We present the related work in this domain, give details about the implementation for clarity and extensibility, and experiment results, investigating the possibility of having an equally efficient version in PyTorch.

5.1 Future Work

In the following works, we suggest an overall average precision performance increase while focusing on small details using higher threshold levels.

The parallelization possibilities of PyTorch can also be explored to increase the efficiency of the network, especially data augmentation.

In this work we could not focus on object tracking, the network can be modified with time context, connecting it to the segmentation masks from the DecoNet and LSTM to use series of data.

Since we did not experiment with the hyperparameters, combining the network with hyperparameter optimization and AutoML algorithms could be another method to increase the performance of the PyTorch model.

6 Acknowledgments

First and foremost, I would like to thank Prof. Brox for agreeing to have multiple meetings with me before I could decide on my thesis topic and for agreeing for me to do the thesis in the Pattern Recognition and Image Processing chair.

I would also like to thank Prof. Teschner for agreeing to be my second examiner on such short notice.

Additionaly, I would like to thank Yassine for always agreeing to have meetings with me in the same week that I ask for and his encouragements when I felt like it was impossible to move forward.

Last but not least, I would like to thank my mother Eda, my father Cihat, and my partner Giota for believing in me and supporting me through the most difficult of the times.

Bibliography

[1] K. He, G. Gkioxari, P. Dollár, and R. B. Girshick, "Mask R-CNN," *CoRR*, vol. abs/1703.06870, 2017.

[2] K. L. S. Vebjorn Ljosa and A. E. Carpenter, "Annotated high-throughput microscopy image sets for validation," *Nature Methods*, vol. 9, no. 7, pp. 637–637, 2012.

[3] A. Böhm, N. Mayer, and T. Brox, "Diskmask: Focusing object features for accurate instance segmentation of elongated or overlapping objects," in *2020 IEEE 17th International Symposium on Biomedical Imaging (ISBI)*, pp. 230–234, 2020.

[4] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," *CoRR*, vol. abs/1505.04597, 2015.

[5] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[6] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, Curran Associates, Inc., 2019.

[7] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *CoRR*, vol. abs/1311.2524, 2013.

[8] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," *CoRR*, vol. abs/1506.01497, 2015.

[9] A. Böhm, M. Tatarchenko, and T. Falk, "Isoo v2 dl - semantic instance segmentation of touching and overlapping objects," pp. 343–347, 04 2019.

[10] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[11] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[12] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[13] S. Van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, and T. Yu, "scikit-image: image processing in python," *PeerJ*, vol. 2, p. e453, 2014.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015.

[15] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[16] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Doll'a r, and C. L. Zitnick, "Microsoft COCO: common objects in context," *CoRR*, vol. abs/1405.0312, 2014.

[17] H. Kumar, "Evaluation metrics for object detection and segmentation: map."