Le génie pour l'industrie

LABORATOIRE D'IMAGERIE, DE VISION ET D'INTELLIGENCE ARTFICIELLE

Summer school on deep learning for medical Imaging 2022

Weakly supervised learning

By José Dolz, Christian Desrosiers, Gustavo Vargas Hakim

Weak labels for image segmentation

We can have weak labels instead of fully labeled images for segmentation.

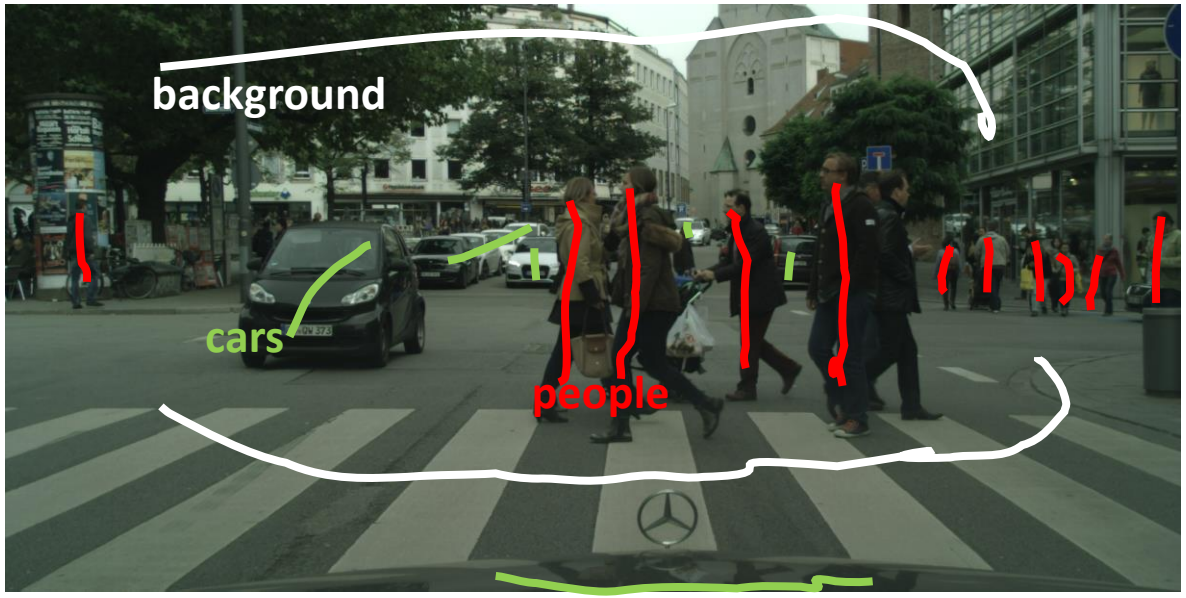


Image from cityscapes dataset (https://www.cityscapes-dataset.com/)

We can still achieve a high-performance segmentation if we choose the **correct priors** to constrain our CNNs.

Segmentation problem

Take an image $\boldsymbol{I}$ as a set of pixels p:

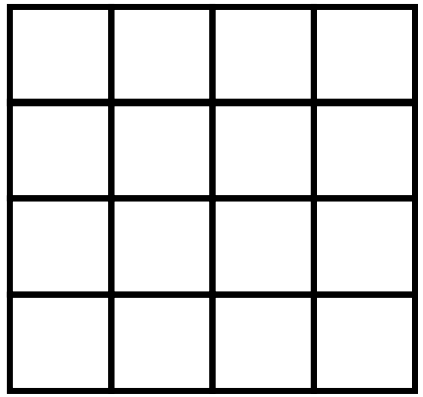


Image $\boldsymbol{I}$

Segmentation problem

Consider Ω as the discrete set of per-pixel labels

Labels Ω

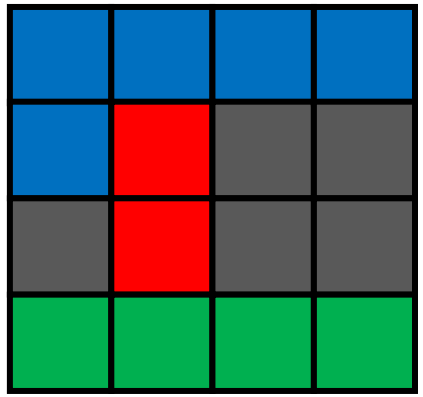


Image I

Segmentation problem

The subset $\Omega_L \subseteq \Omega$ contains the weak labels of image $\boldsymbol{I}$, which means only a few pixels per class are labeled:

Weak Labels Ω_L

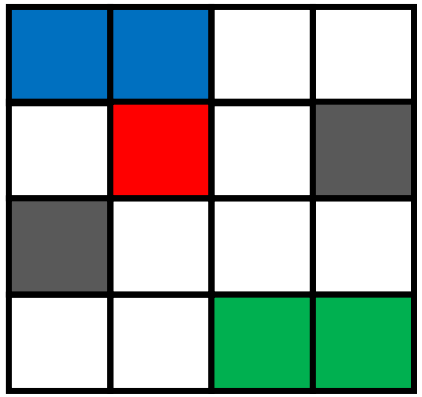


Image $\boldsymbol{I}$

Segmentation problem

The predictions of the network are made per-pixel. Each pixel p has its own vector of softmax probabilities $\boldsymbol{S}_p$:

Weak Labels Ω_L

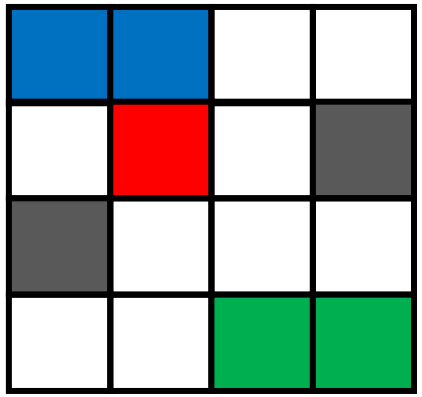
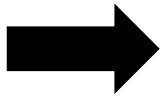


Image $\boldsymbol{I}$



$\boldsymbol{S}_1$	$\boldsymbol{S}_2$	$\boldsymbol{S}_3$	$\boldsymbol{S}_4$
$\boldsymbol{S}_5$	$\boldsymbol{S}_6$	$\boldsymbol{S}_7$	$\boldsymbol{S}_8$
$\boldsymbol{S}_9$	$\boldsymbol{S}_{10}$	$\boldsymbol{S}_{11}$	$\boldsymbol{S}_{12}$
$\boldsymbol{S}_{13}$	$\boldsymbol{S}_{14}$	$\boldsymbol{S}_{15}$	$\boldsymbol{S}_{16}$

Probability vectors $\boldsymbol{S}$

$\boldsymbol{S}_p = [0.1, 0.2, 0.6, 0.1]$

One probability per class

Segmentation problem

Under the fully-supervised setting, we calculate the cross-entropy loss function as follows:

Labels Ω

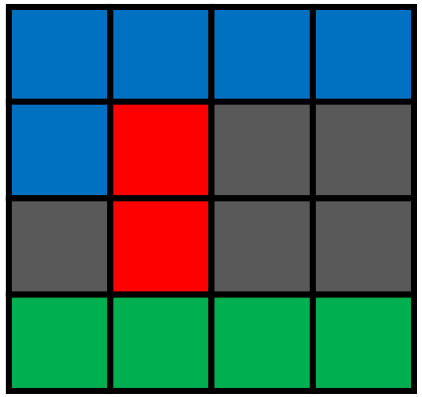


Image $\boldsymbol{I}$

$$\mathcal{H}(\boldsymbol{S}) = -\sum_{p \in \Omega} \log \boldsymbol{S}_p$$

Segmentation problem

Under the weakly-supervised setting, we can utilize the partial cross-entropy loss. We now utilize the weakly-labeled set Ω_L:

Weak Labels Ω_L

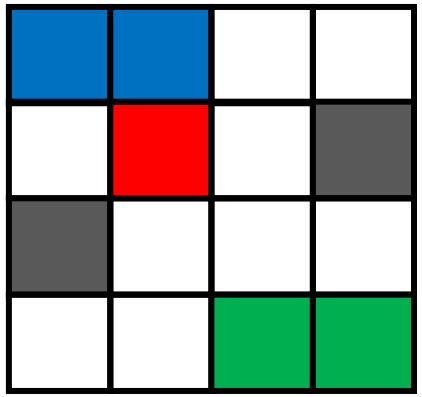


Image $\boldsymbol{I}$

$$\mathcal{H}(\boldsymbol{S}) = -\sum_{p \in \Omega_L} \log \boldsymbol{S}_p$$

Adding penalties

We want to optimize a loss function that is subject to some constraints. This problem might be difficult to solver, so we can transform the constrains into penalty functions.

Network parameters

Loss function (e.g. cross-entropy)

$$\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$$

$$subject\ to \quad f_1(s_{\boldsymbol{\theta}}^n) \leq 0, n = 1, \dots, N$$
$$\vdots$$
$$f_P(s_{\boldsymbol{\theta}}^n) \leq 0, n = 1, \dots, N$$

Inequality constraints

The constraints are applied on the softmax predictions. For example, specifying that the segmented regions per class do not surpass a certain limit.

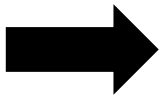
Adding penalties

The penalty function is designed in such a way that it still corresponds to the original constraints but penalizing the original loss function when the solutions do not fit the restrictions. Now we have an unconstrained problem, which can be easier to optimize:

$$\min_{\boldsymbol{\theta}} \boldsymbol{\mathcal{L}}(\boldsymbol{\theta})$$

$$subject\ to \quad \begin{matrix} f_1(s_{\boldsymbol{\theta}}^n) \leq 0, n = 1, \dots, N \\ \vdots \\ f_P(s_{\boldsymbol{\theta}}^n) \leq 0, n = 1, \dots, N \end{matrix}$$

Inequality constraints



$$\min_{\boldsymbol{\theta}} \boldsymbol{\mathcal{L}}(\boldsymbol{\theta}) + \sum_{i=1}^{P} f_i(s_{\boldsymbol{\theta}}^n)$$

Unconstrained problem

But what kind of inequality constraints can help us with weakly-supervised segmentation?

Using size priors

Let’s propose size priors; if we know that the regions in the image most be bounded within certain size, we can enforce the network to segment within that range. Here we use **size loss** as a penalty that corresponds to this size constraint:

Weak Labels Ω_L

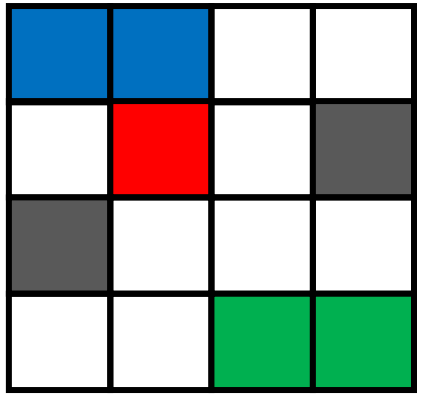


Image $\boldsymbol{I}$

Constant value

$$\mathcal{H}(\boldsymbol{S}) + \lambda C(V_s)$$

Partial cross-entropy

$$V_s = \sum_{p \in \Omega} \boldsymbol{S}_p$$

Kervadec, H., Dolz, J., Tang, M., Granger, E., Boykov, Y., Ben Ayed, I. (2019). *Constrained-CNN losses for weakly supervised segmentation.* Medical Image Analysis 54, p. 888-899

Using size priors

Let’s propose size priors; if we know that the regions in the image most be bounded within certain size, we can enforce the network to segment within that range. Here we use **size loss**:

Weak Labels Ω_L

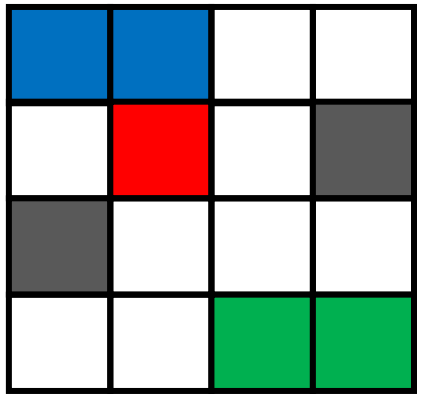


Image $\boldsymbol{I}$

$$\mathcal{H}(\boldsymbol{S}) + \lambda C(V_s)$$

Size Lower bound

$$C(V_s) = \begin{cases} (V_s - a)^2, if \;\; V_s < a \\ (b - V_s)^2, if \;\; V_s > b \\ 0, \qquad otherwise \end{cases}$$

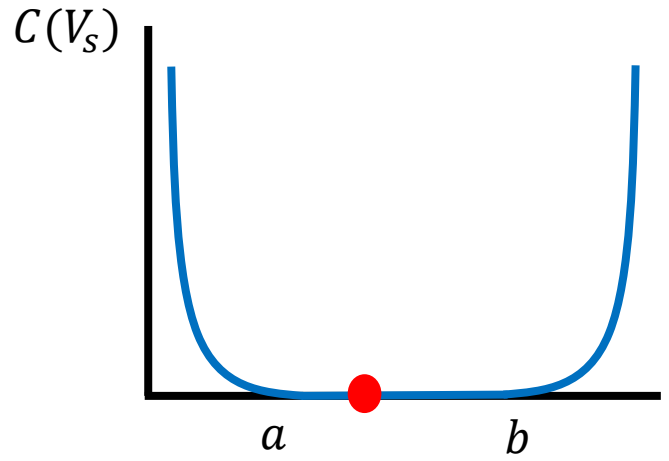
Size Upper bound

Kervadec, H., Dolz, J., Tang, M., Granger, E., Boykov, Y., Ben Ayed, I. (2019). *Constrained-CNN losses for weakly supervised segmentation.* Medical Image Analysis 54, p. 888-899

Using size priors

When we use **backpropagation**, we can calculate the derivative of the size loss as follows:

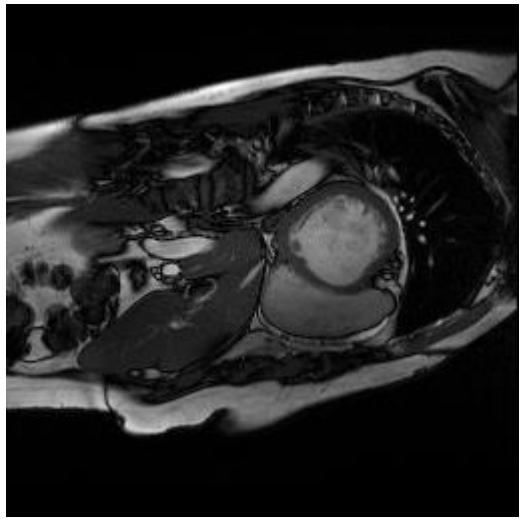
$$-\frac{\partial C(V_s)}{\partial \boldsymbol{\theta}} \propto \begin{cases} (a - V_s)\dfrac{\partial S_p}{\partial \boldsymbol{\theta}}, & if\ \ V_s < a \\ (b - V_s)\dfrac{\partial S_p}{\partial \boldsymbol{\theta}}, & if\ \ V_s > b \\ 0, & otherwise \end{cases}$$



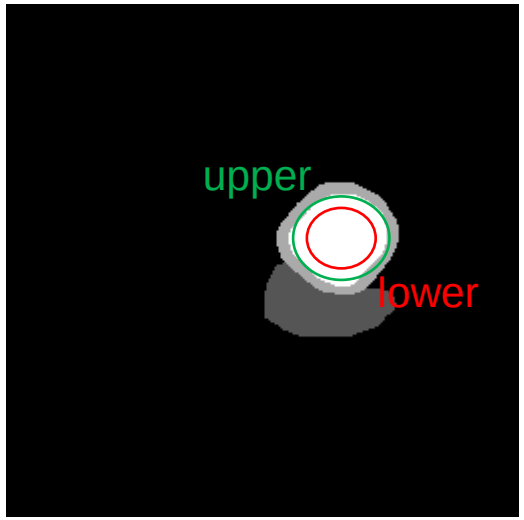
Kervadec, H., Dolz, J., Tang, M., Granger, E., Boykov, Y., Ben Ayed, I. (2019). *Constrained-CNN losses for weakly supervised segmentation.* Medical Image Analysis 54, p. 888-99

Using size priors

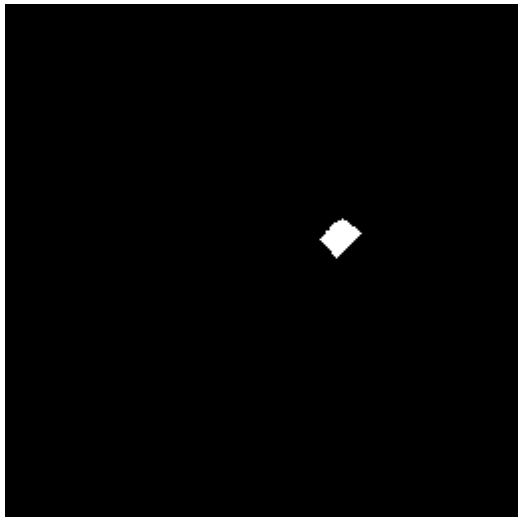
We can now apply this constraint in medical images. Here, an expert can specify the lower and upper bounds of the size of the region that we need to segment. Using these constraints, and having only a small segmented region, we can still obtain good segmentation results.



Original



Full labels Ω



Weak labels Ω_L

Images from ACDC dataset (https://www.creatis.insa-lyon.fr/Challenge/acdc/databases.html)

Using log-barriers extensions

The log-barriers is a method to transform this constrained optimization problem into an unconstrained optimization problem. However, we need to start from a feasible solution, which is not trivial to find for a deep network. We then can use the **log-barrier extensions**:

Loss function (e.g. cross-entropy)

Log-barrier extension

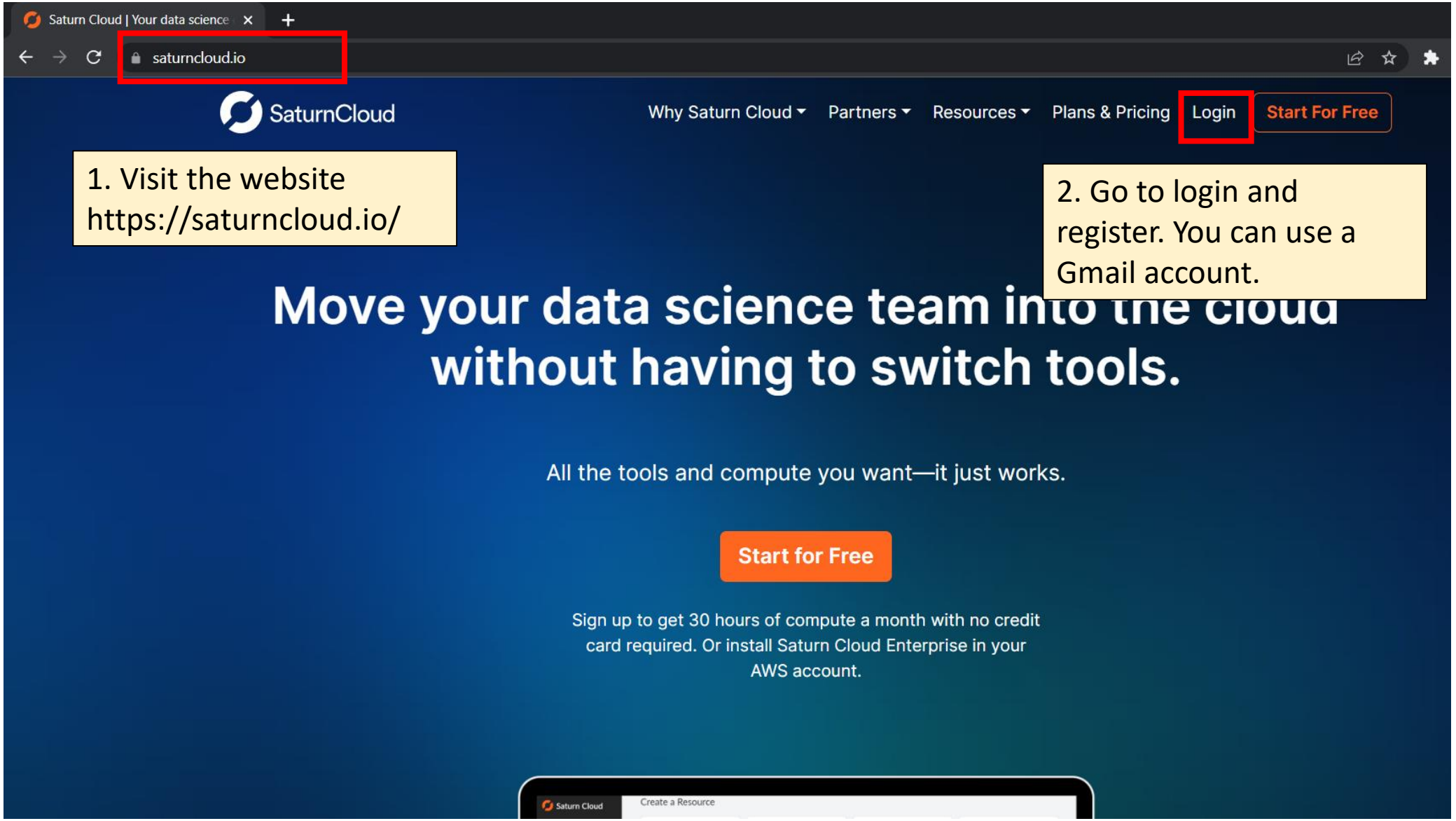
$$\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) + \sum_{i=1}^{P} \sum_{n=1}^{N} \tilde{\varphi}_t\left(f_i(s_{\boldsymbol{\theta}}^n)\right)$$

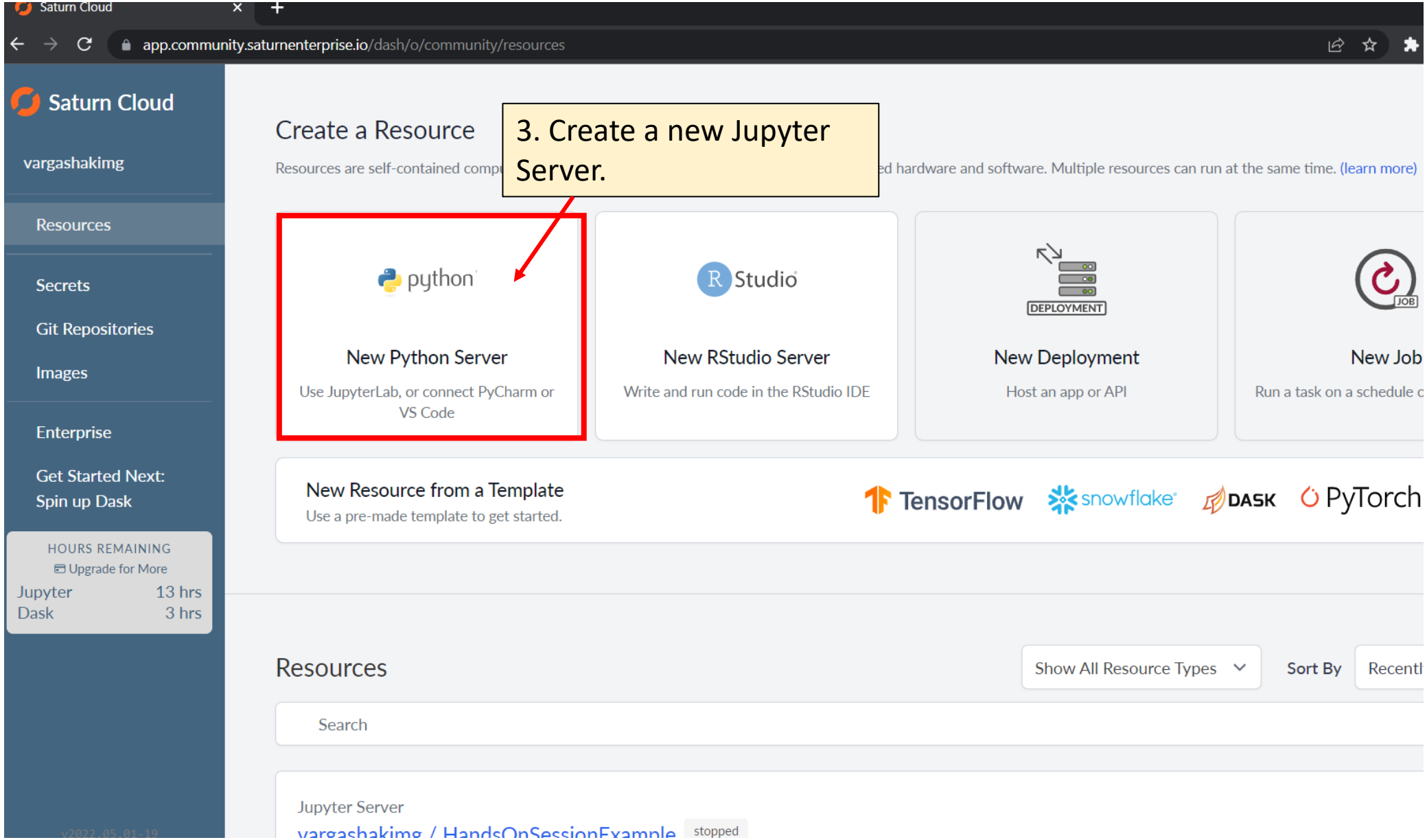
Network parameters

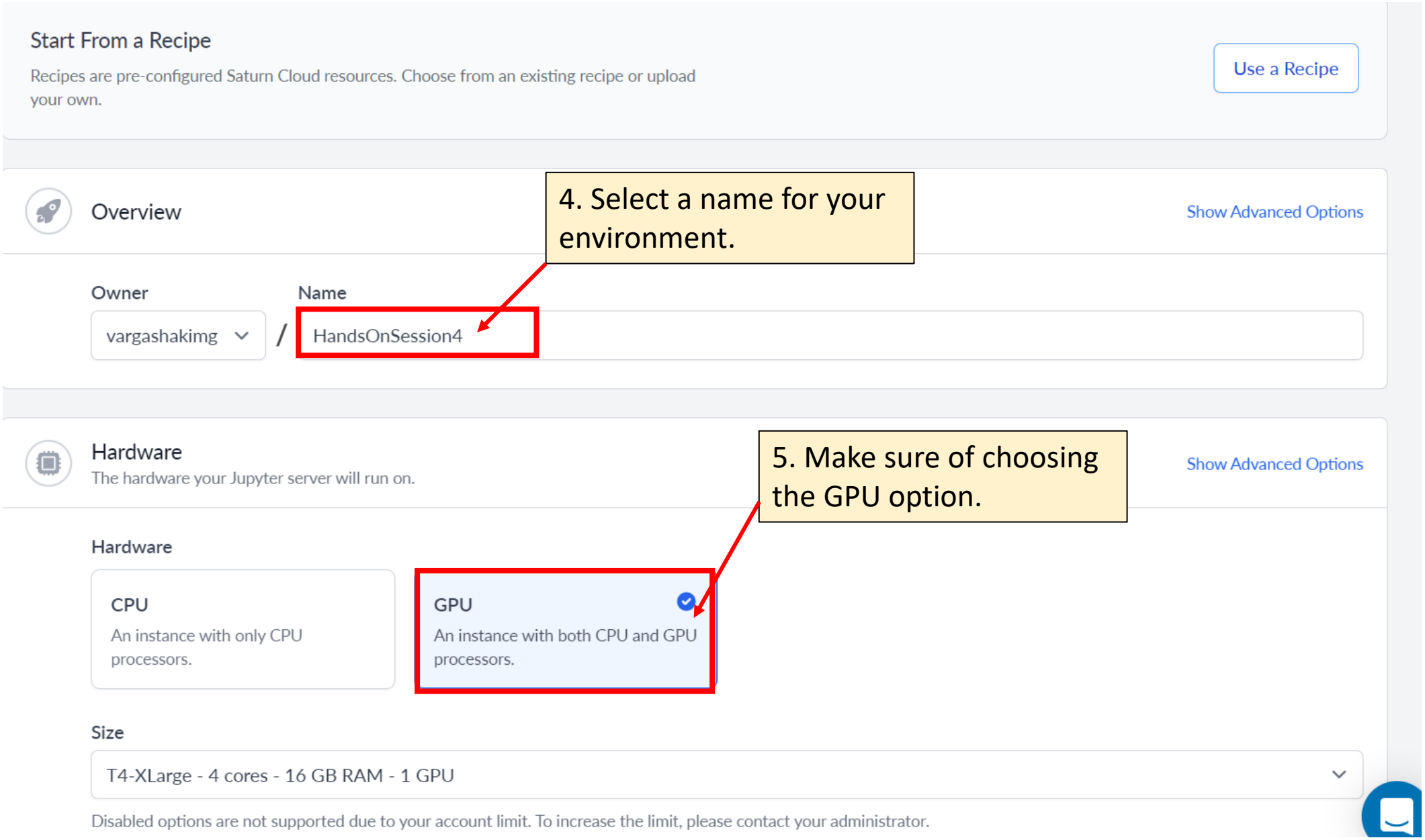
$$\tilde{\varphi}_t(\mathrm{z}) = \begin{cases} -\frac{1}{t}\log(-z), & if\ z \leq -\frac{1}{t^2} \\ tz - \frac{1}{t}\log\left(\frac{1}{t^2}\right) + \frac{1}{t}, & otherwise \end{cases}$$

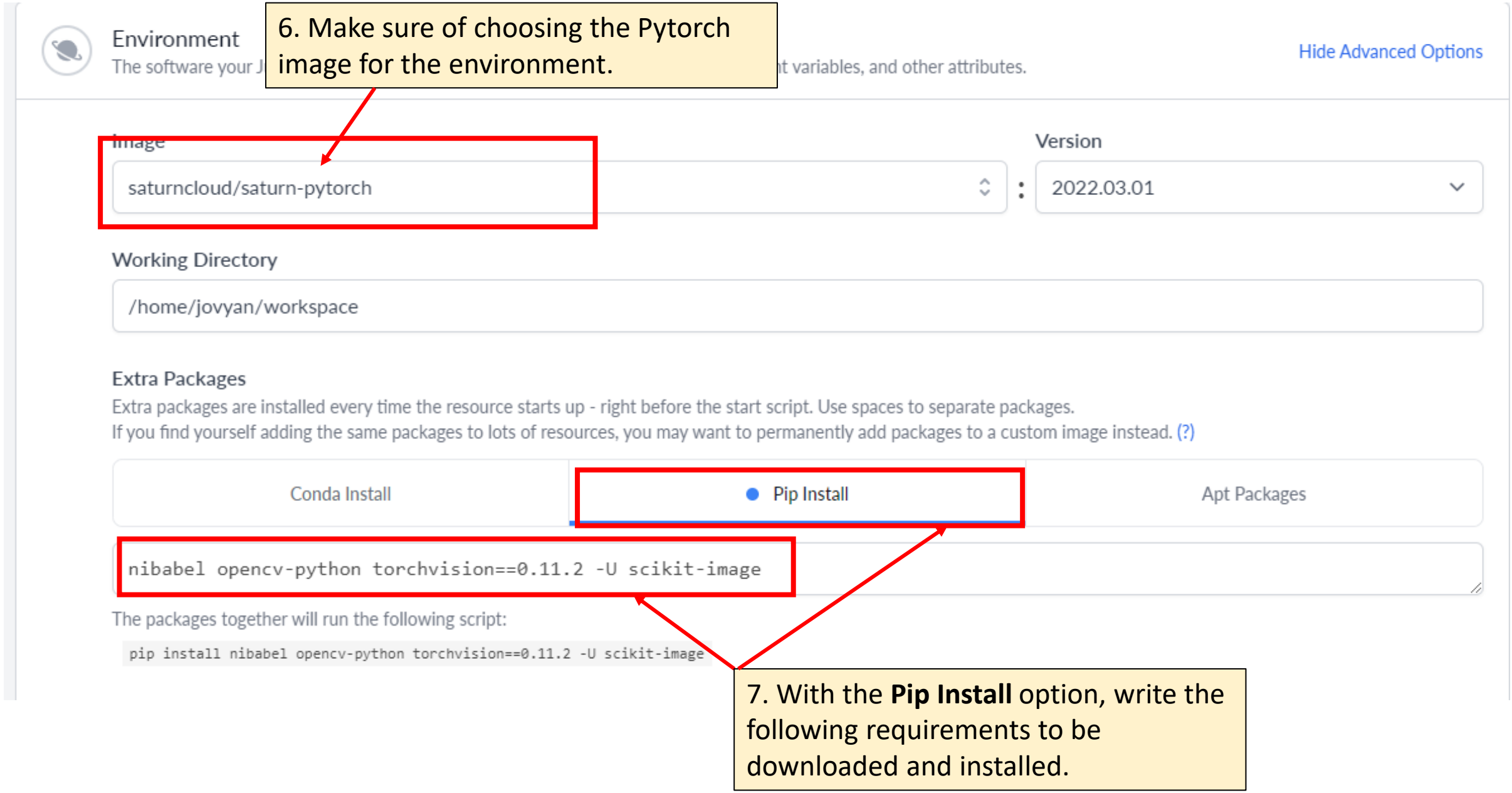
Kervadec, H., Dolz, J., Yuan, J., Desrosiers, C., Granger, E., Ben Ayed, I. (2022). *Constrained deep networks: Lagrangian optimization via Log-barrier extensions.* ArXiv Preprint.

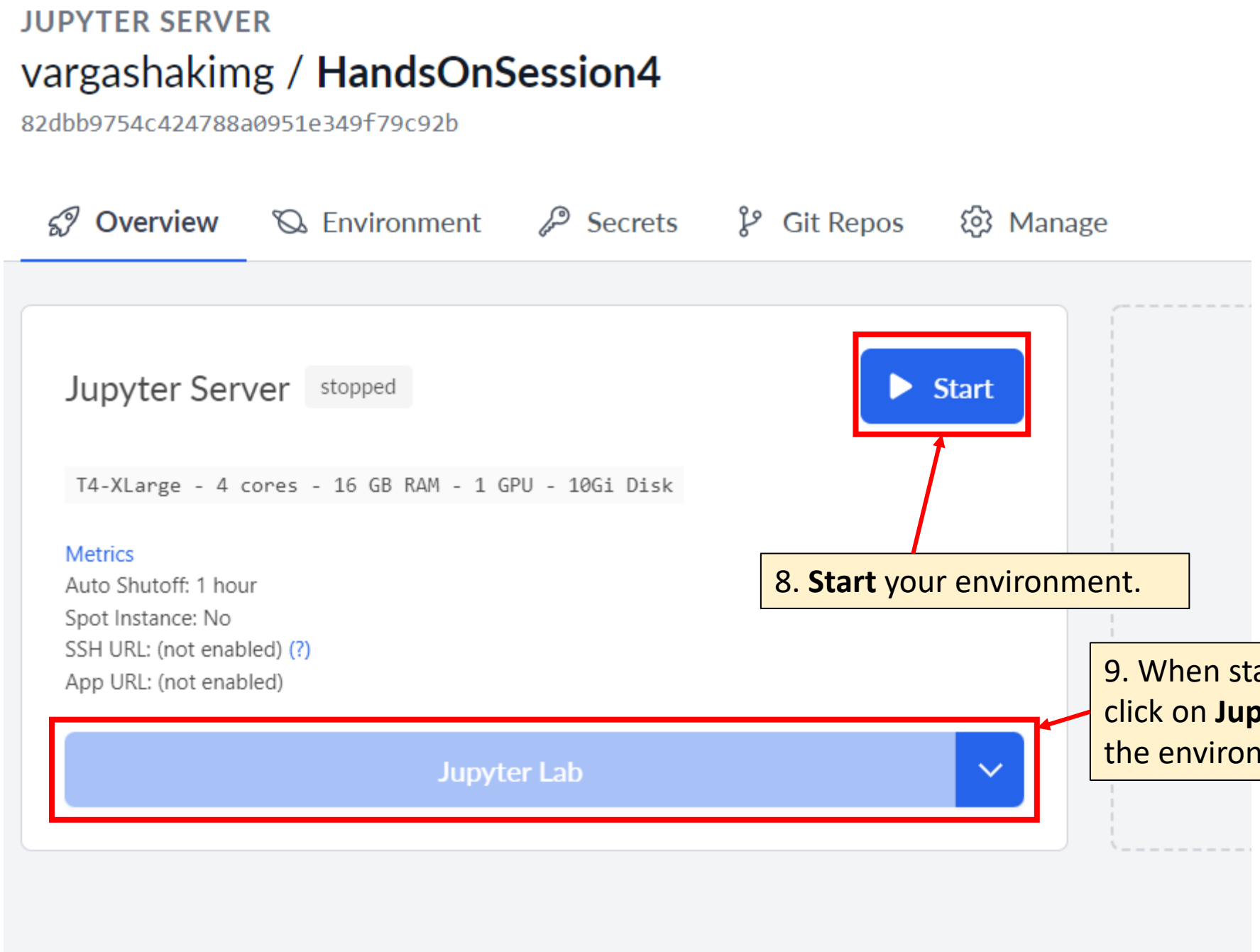
Configuring SaturnCloud environment


Saturn Cloud | Your data science
saturncloud.io
SaturnCloud
Why Saturn Cloud
Partners
Resources
Plans & Pricing
Login
Start For Free
1. Visit the website https://saturncloud.io/
2. Go to login and register. You can use a Gmail account.
Move your data science team into the cloud without having to switch tools.
All the tools and compute you want—it just works.
Start for Free
Sign up to get 30 hours of compute a month with no credit card required. Or install Saturn Cloud Enterprise in your AWS account.
Saturn Cloud
Create a Resource

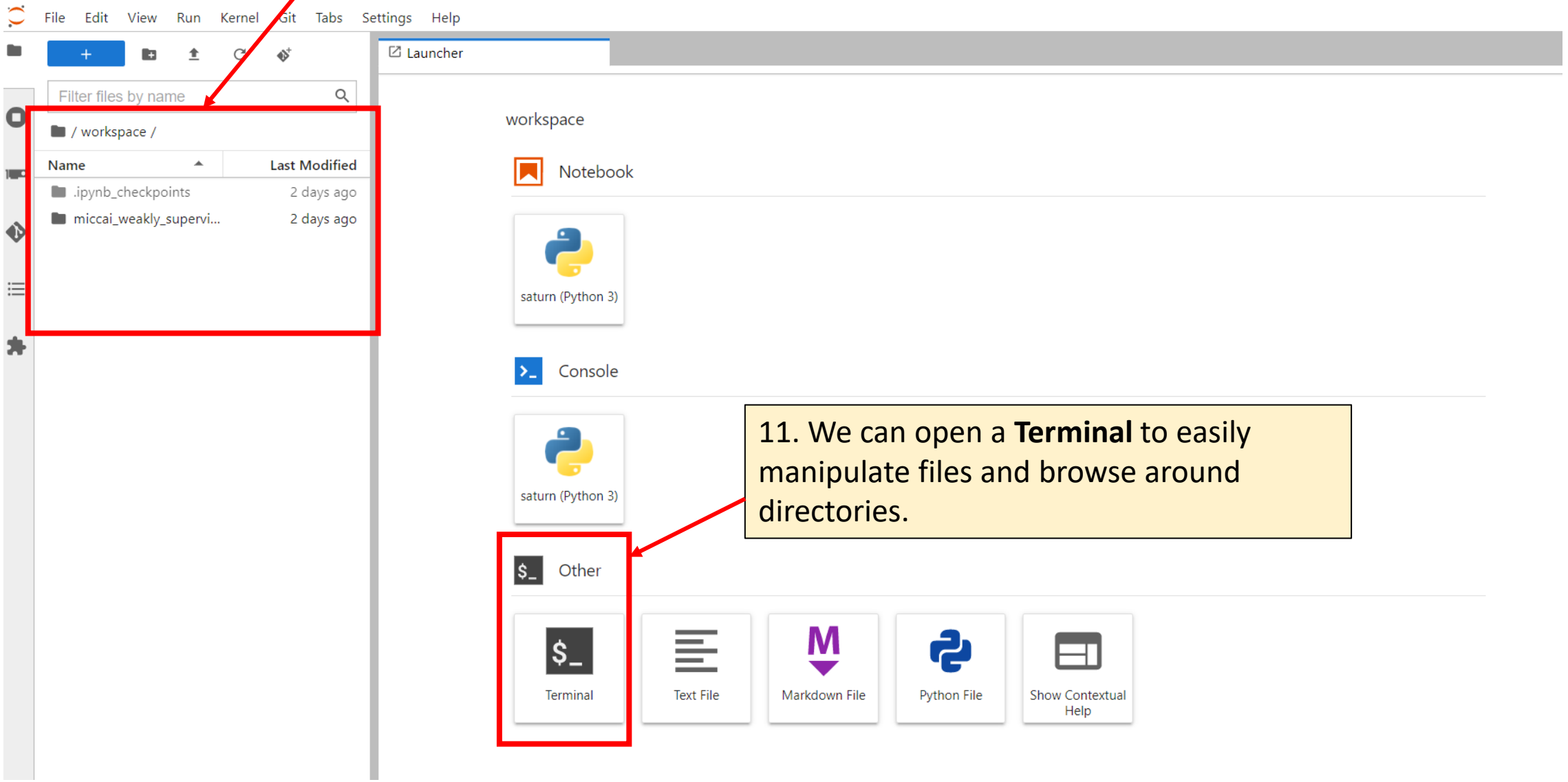

Saturn Cloud
app.community.saturnenterprise.io/dash/o/community/resources
Saturn Cloud
vargashakimg
Resources
Secrets
Git Repositories
Images
Enterprise
Get Started Next:
Spin up Dask
HOURS REMAINING
Upgrade for More
Jupyter 13 hrs
Dask 3 hrs
Create a Resource
3. Create a new Jupyter Server.
Resources are self-contained compu
ed hardware and software. Multiple resources can run at the same time. (learn more)
python
New Python Server
Use JupyterLab, or connect PyCharm or VS Code
R Studio
New RStudio Server
Write and run code in the RStudio IDE
DEPLOYMENT
New Deployment
Host an app or API
JOB
New Job
Run a task on a schedule c
New Resource from a Template
Use a pre-made template to get started.
TensorFlow
snowflake
DASK
PyTorch
Resources
Show All Resource Types
Sort By
Recentl
Search
Jupyter Server
vargashakimg / HandsOnSessionExample
stopped


Start From a Recipe
Recipes are pre-configured Saturn Cloud resources. Choose from an existing recipe or upload your own.
Use a Recipe
Overview
4. Select a name for your environment.
Show Advanced Options
Owner
Name
vargashakimg
/
HandsOnSession4
Hardware
The hardware your Jupyter server will run on.
5. Make sure of choosing the GPU option.
Show Advanced Options
Hardware
CPU
An instance with only CPU processors.
GPU
An instance with both CPU and GPU processors.
Size
T4-XLarge - 4 cores - 16 GB RAM - 1 GPU
Disabled options are not supported due to your account limit. To increase the limit, please contact your administrator.


Environment
The software your J
t variables, and other attributes.
Hide Advanced Options
6. Make sure of choosing the Pytorch image for the environment.
Image
saturncloud/saturn-pytorch
Version
2022.03.01
Working Directory
/home/jovyan/workspace
Extra Packages
Extra packages are installed every time the resource starts up - right before the start script. Use spaces to separate packages.
If you find yourself adding the same packages to lots of resources, you may want to permanently add packages to a custom image instead. (?)
Conda Install
Pip Install
Apt Packages
nibabel opencv-python torchvision==0.11.2 -U scikit-image
The packages together will run the following script:
pip install nibabel opencv-python torchvision==0.11.2 -U scikit-image
7. With the **Pip Install** option, write the following requirements to be downloaded and installed.


JUPYTER SERVER
vargashakimg / HandsOnSession4
82dbb9754c424788a0951e349f79c92b
Overview
Environment
Secrets
Git Repos
Manage
Jupyter Server
stopped
Start
T4-XLarge - 4 cores - 16 GB RAM - 1 GPU - 10Gi Disk
Metrics
Auto Shutoff: 1 hour
Spot Instance: No
SSH URL: (not enabled) (?)
App URL: (not enabled)
Jupyter Lab
8. Start your environment.
9. When starting finishes, click on Jupyter Lab to open the environment.

10. The **workspace** section is where all our files and code is going to be located.

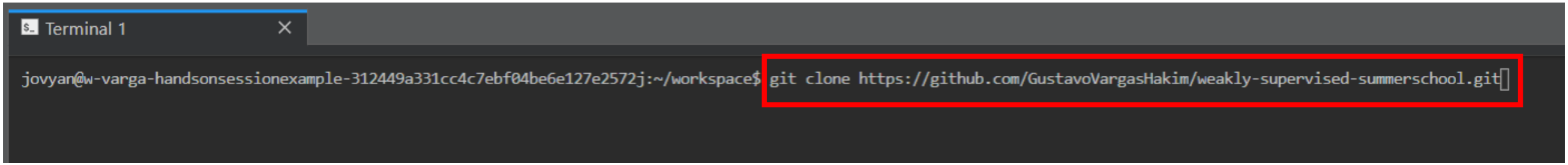


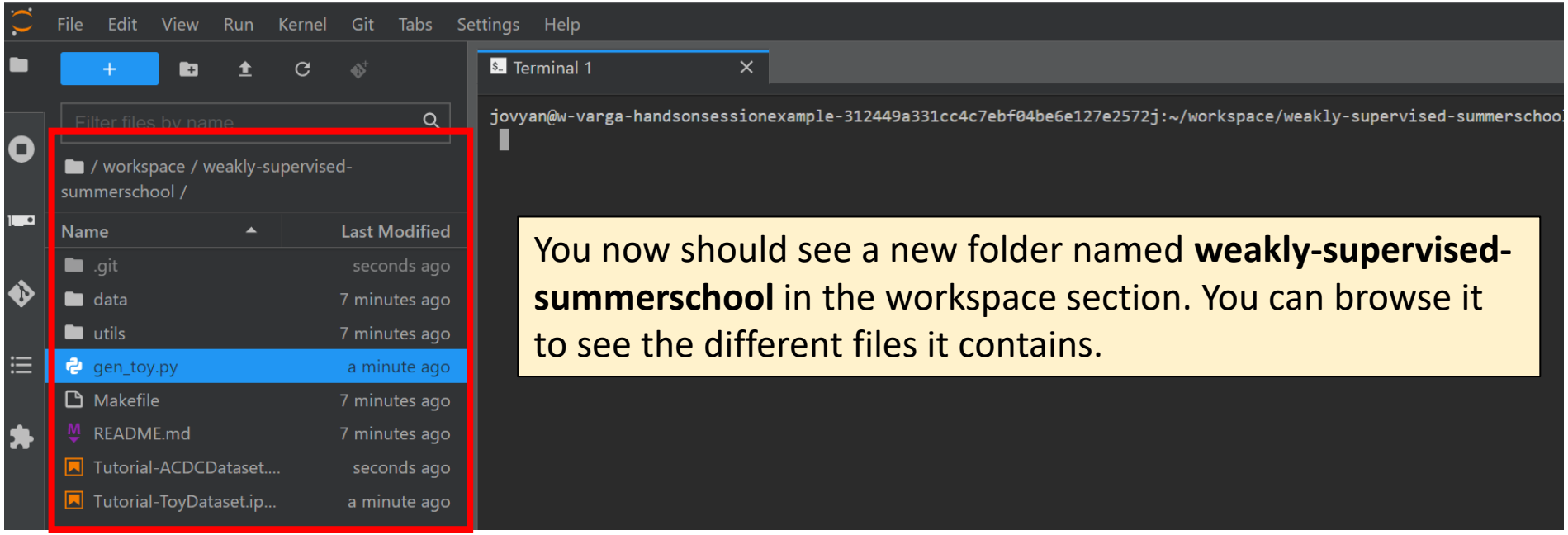
11. We can open a **Terminal** to easily manipulate files and browse around directories.

Preparing the code and data

Click on **Terminal** to open a new a new terminal so that we can download the repository for this tutorial. The command to use is:
git clone https://github.com/GustavoVargasHakim/weakly-supervised-summerschool.git

Press **enter** afterwards. The repository will start downloading and will be visible from the workspace section.

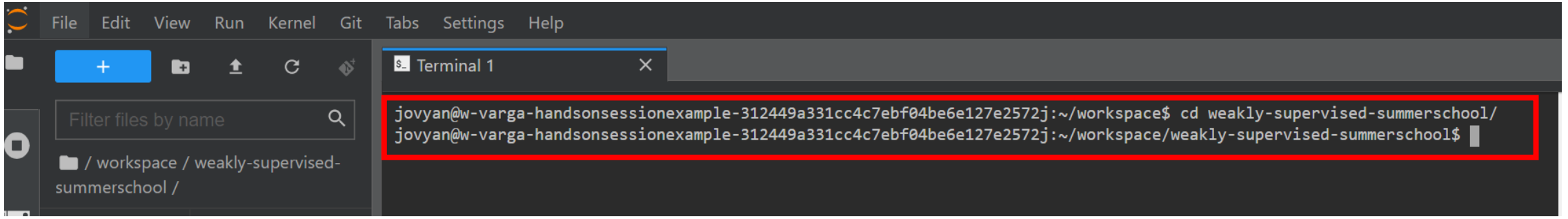




You now should see a new folder named **weakly-supervised-summerschool** in the workspace section. You can browse it to see the different files it contains.

Now we will create the toy dataset for the first part of this tutorial. First, let's move to the repository folder from the terminal. Use the command:

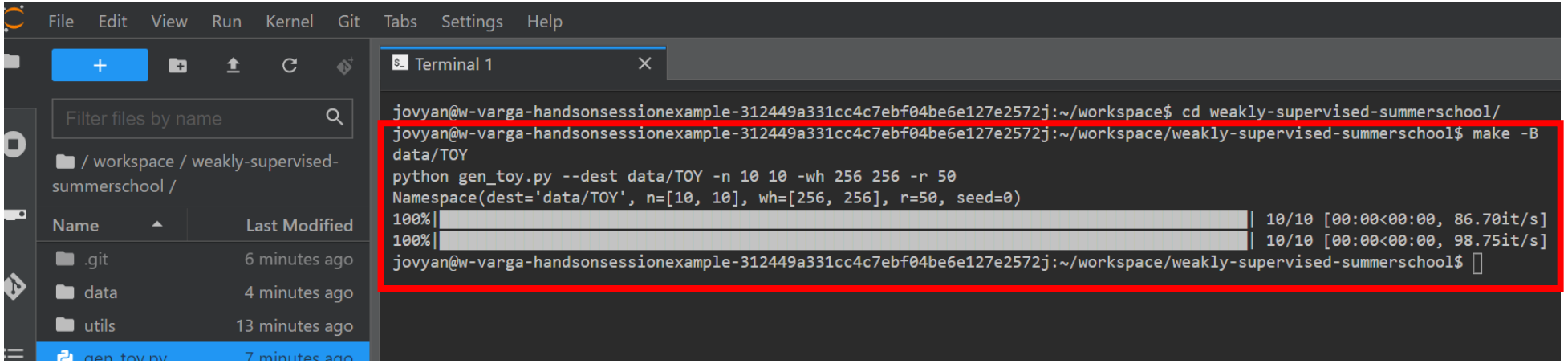
cd weakly-supervised-summerschool/



To create the first dataset, we use the following command in the terminal:

make -B data/TOY

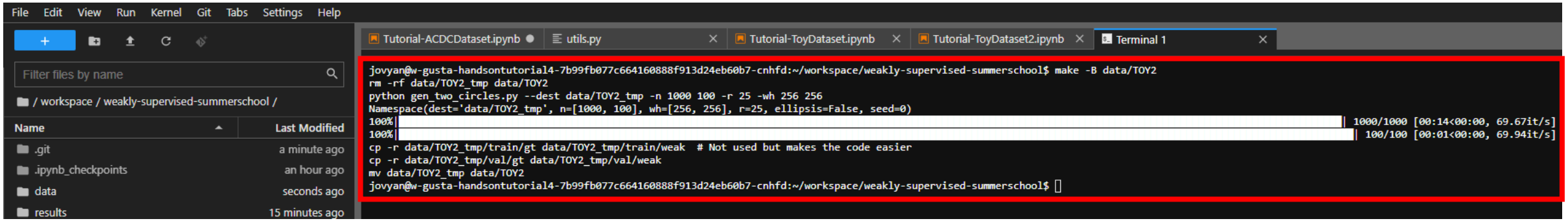
This will create a folder called **TOY** inside the folder **data**, where the images will be placed.



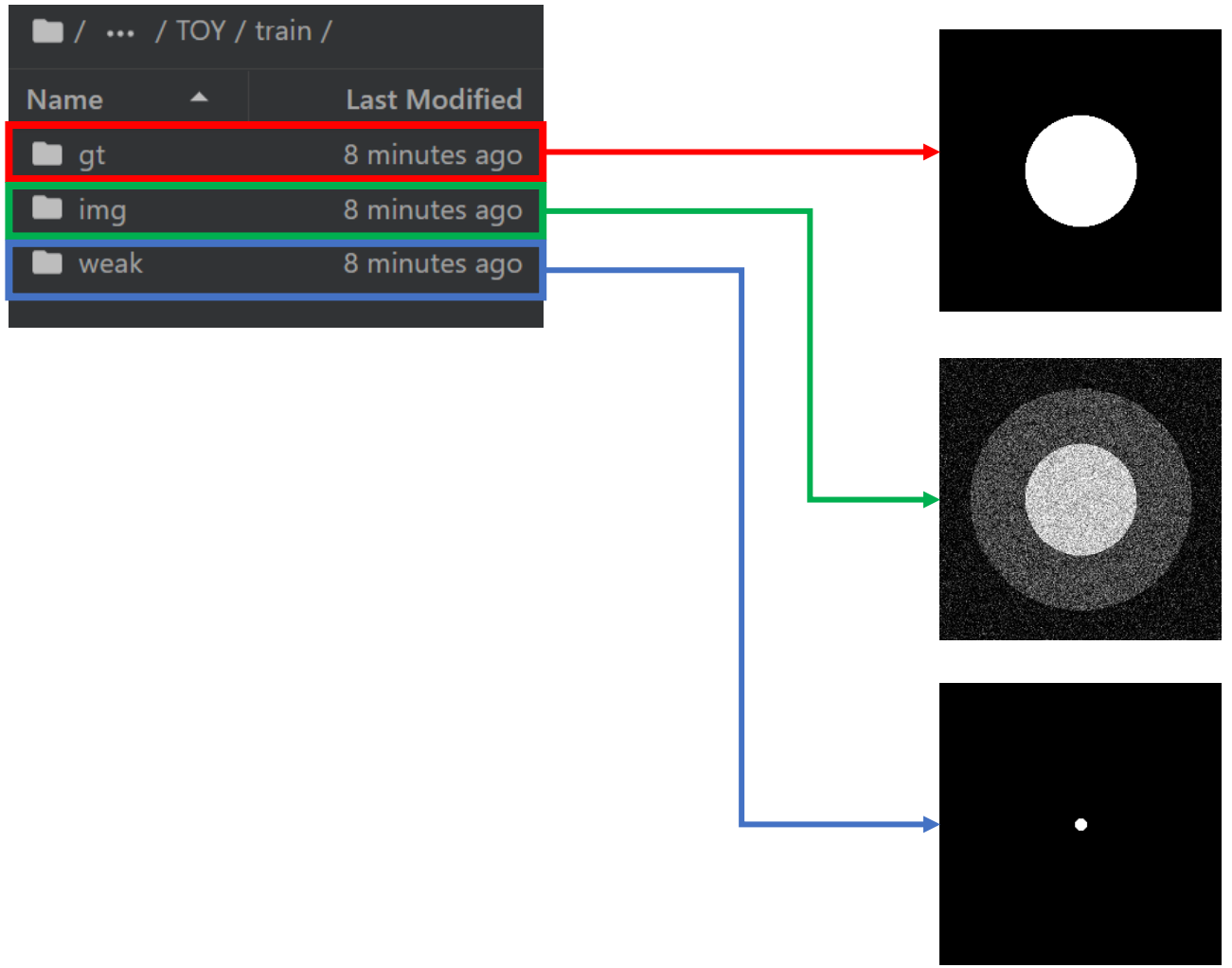
To create the second dataset, we use the following command in the terminal:

make -B data/TOY2

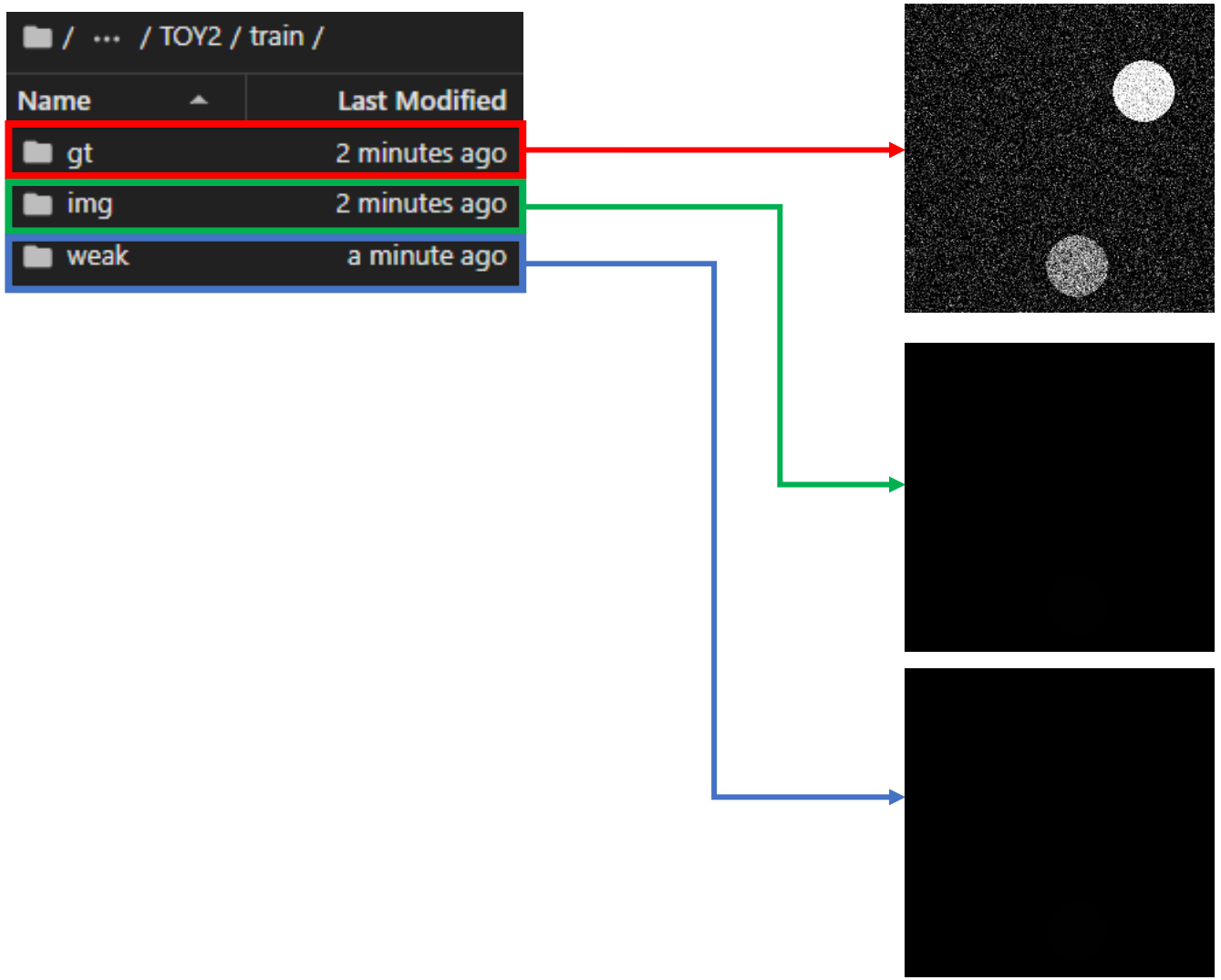
This will create a folder called **TOY2** inside the folder **data**, where the images will be placed.



Inside the **data/TOY** folder on the workspace section, we see two subfolders, called **train** and **val**, each containing the training and validation data, respectively. If we take a look at any of them, we will find three subfolders, called **gt** for the ground truths (full labels), **img** for the images, and **weak** for the weak (partial) labels.



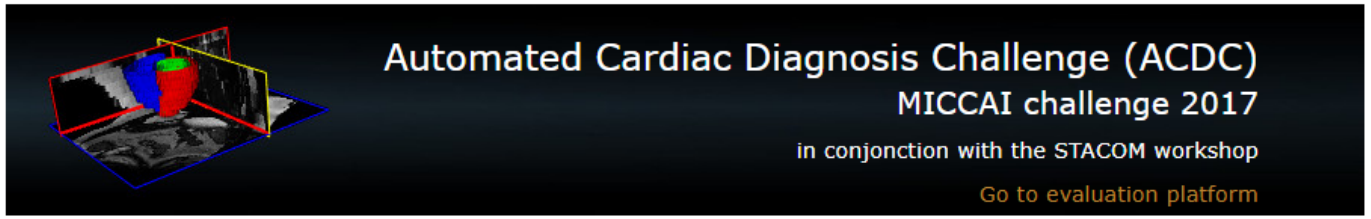
Similarly, inside the **data/TOY2** folder on the workspace section, we see two subfolders, called **train** and **val**, each containing the training and validation data, respectively. If we take a look at any of them, we will find three subfolders, called **gt** for the ground truths (full labels), **img** for the images, and **weak** for the weak (partial) labels.



The Automated Cardiac Diagnosis Challenge (ACDC) dataset might take a little longer to download and configure. To do so, please go to the website:

https://www.creatis.insa-lyon.fr/Challenge/acdc/databases.html

Then select the tab called **Training dataset** and then click on the link that says "…the online evaluation plateform." at the end of the text.



- Overview
- Participation
- Databases
- Evaluation
- Code
- MICCAI'17 results
- Contact

Databases

Overview | Training dataset | Testing dataset | Classification rules

General description

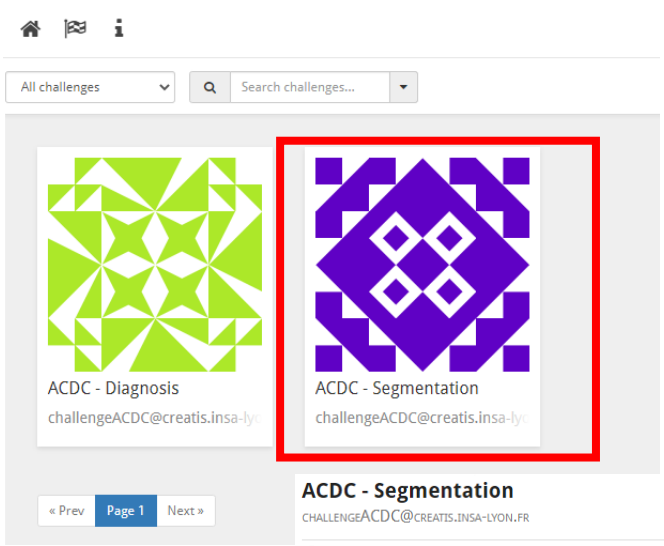
The training database is composed of 100 patients as follows:

- 20 healthy patients;
- 20 patients with previous myocardial infarction;
- 20 patients with dilated cardiomyopathy;
- 20 patients with an hypertrophic cardiomyopathy;
- 20 patients with abnormal right ventricle;

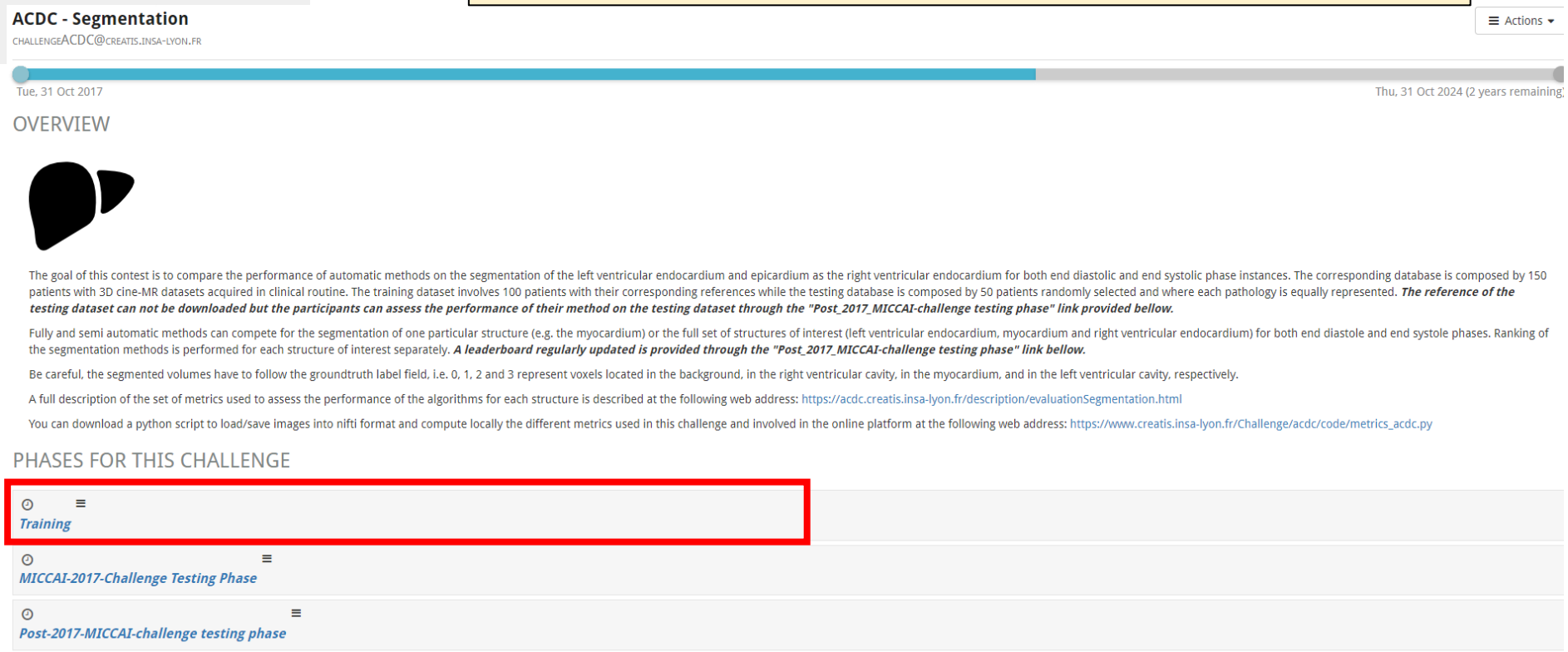
For all these data, the corresponding manual references given by one clinical expert along with additional information on the patient (age, weight, height and diastolic-systolic phase instants) are also provided to the participants.

You may download the training dataset (image + groundtruth) by registring on the online evaluation plateform.

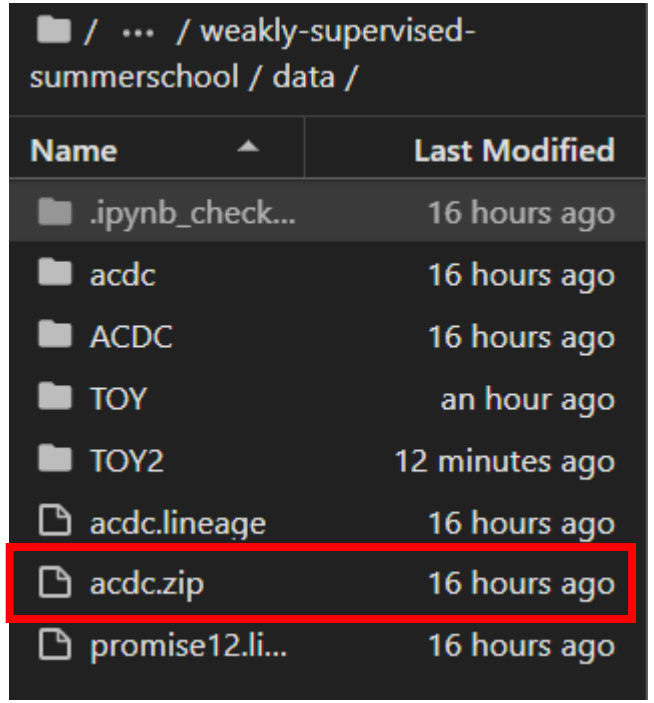
In the next webpage, we need to choose the option named **ACDC – Segmentation**. Following, we click on the tab **Training**.



The website will ask us to register in order to download the dataset. The file to download is .zip file. When the download is finished, make sure of changing the name of the file to **acdc.zip**.



The download of the ACDC dataset might take a few minutes. When it is ready, you can simply drag the file to the folder **data** in the workspace of Jupyter Lab. This might also take a few minutes. **You can continue with the other parts of the tutorial in the meantime.**

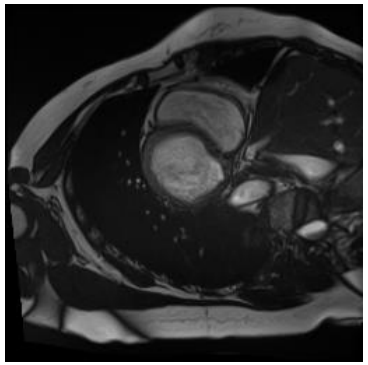


When the file is already loaded in the workspace, we can go the terminal and enter the following commands:

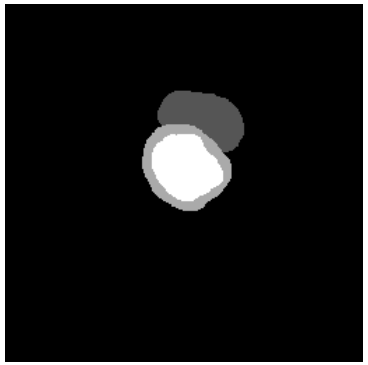
sudo apt-get install unzip
make -B data/ACDC

This will create a new folder called **ACDC** with a very similar structure as in **TOY** and **TOY2**. We can now see the images (img), the ground truths (gt) and the weak masks (weak):

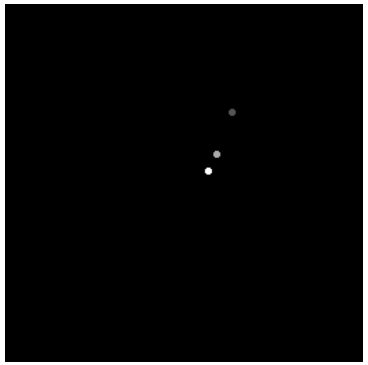
Image



Ground truth

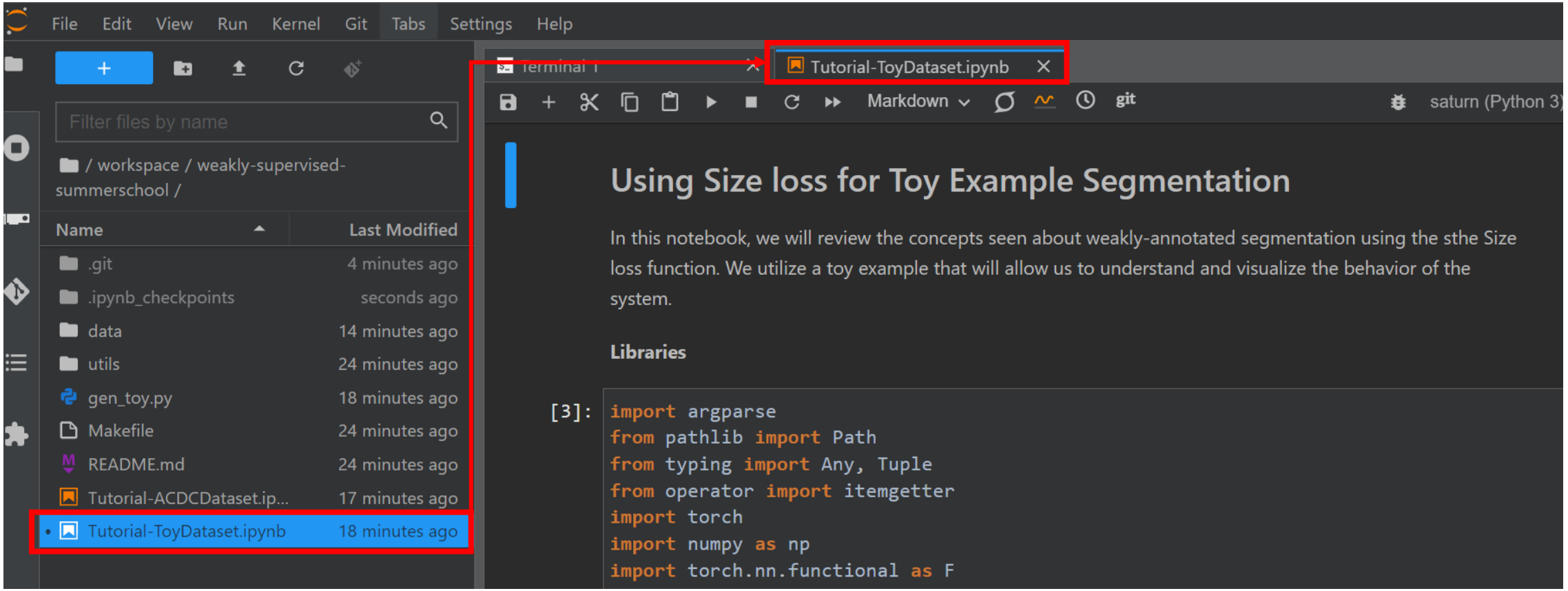


Weak mask

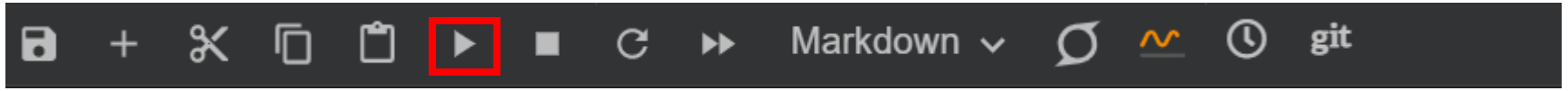


Weakly supervised segmentation:
Toy dataset 1

Back in the **weakly-supervised-summerschool** folder (root folder), we will open the Jupyter notebook called **Tutorial-ToyDataset.ipynb**. This notebook contains the code we need to run the weakly supervised segmentation method of this tutorial on the toy dataset.



If not familiar with Jupyter Notebooks, you can see that the code is separated in cells that we can execute independently. We can do so by pressing the **play** button on top of the notebook, or by using the shortcut **alt + enter/option + enter**, depending if you are a windows/linux user, or a macOS user.



The first two cells import the building blocks we need for our code. The first cell imports python libraries, such as PyTorch and Numpy. The second cell imports pre-defined code files that are available mainly in the folder called **utils**.

Terminal 1 | Tutorial-ToyDataset.ipynb

Code | git

Libraries

[3]:
```
import argparse
from pathlib import Path
from typing import Any, Tuple
from operator import itemgetter
import torch
import numpy as np
import torch.nn.functional as F
from torch import nn, einsum
from torchvision import transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
```
Last executed at 2022-06-08 16:04:59 in 5ms

Utilities

[4]:
```
from utils.dataset import (SliceDataset)
from utils.residual_unet import ResidualUNet
from utils.losses import (CrossEntropy,
                          PartialCrossEntropy,
                          NaiveSizeLoss)
from utils.utils import (weights_init,
                         saveImages,
                         class2one_hot,
                         probs2one_hot,
                         one_hot,
                         tqdm_,
                         dice_coef)
```
Last executed at 2022-06-08 16:05:01 in 182ms

We **run the two cells** to ensure that all these files and libraries are accessible by the code at all times from now on.

The following cell will configure the GPU we are going to use and save it under the variable **device**.

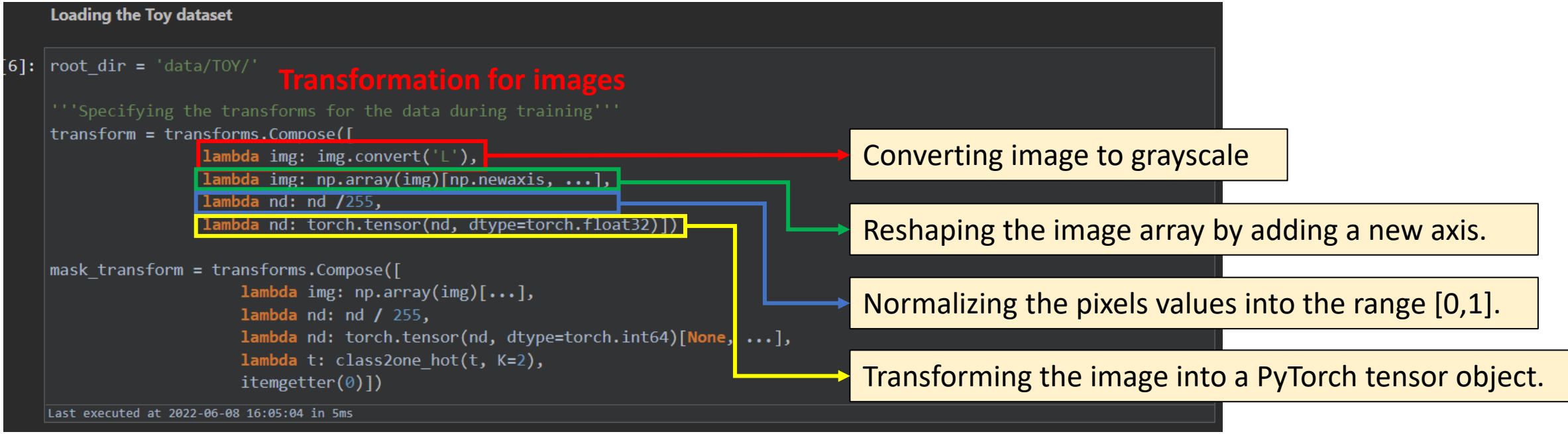
Configuring GPU

If there is an available GPU, it is going to be contained in the variable 'device', otherwise it is going to be simply the GPU.

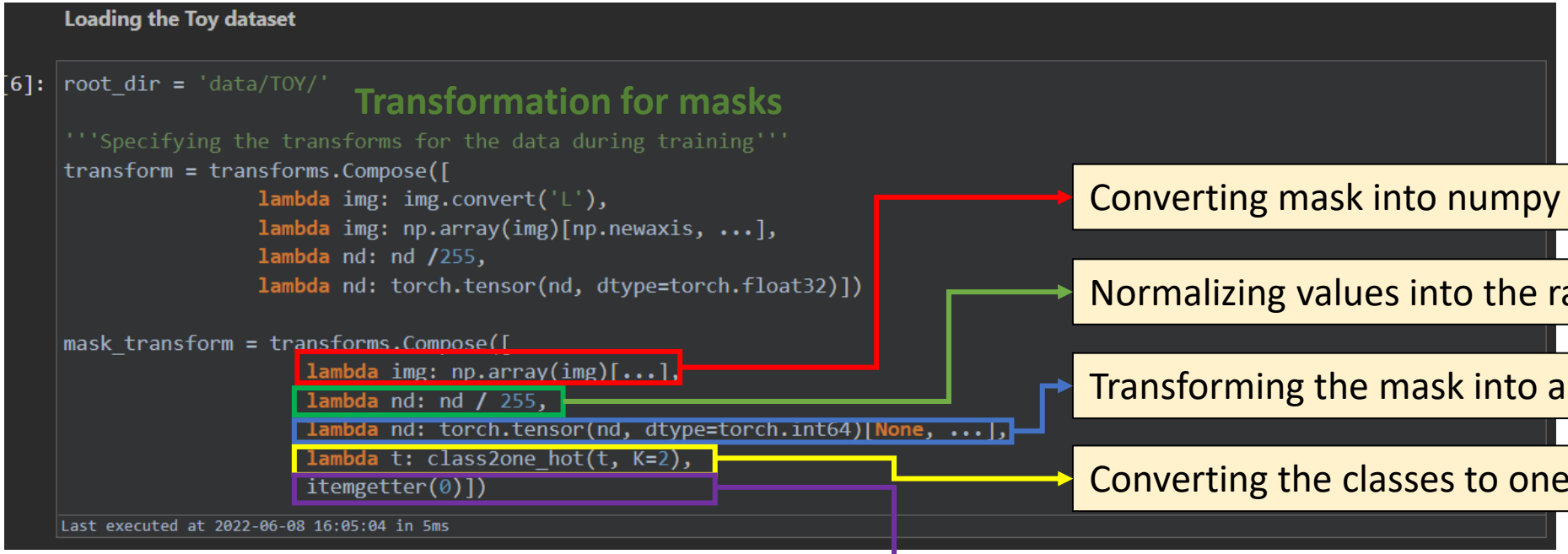
[5]:
```
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
```
Last executed at 2022-06-08 16:05:02 in 29ms

This will only work if the configuration of the environment was correct and the GPU was selected. Otherwise, the variable **device** will contain the CPU.

When we load image data, we oftentimes need to transform it to be accessible during training. For this reason, we can define a series of transformations that are going to be applied one after the other so that all the images are in the correct formatting to be used to train our CNN. First, we **transform the images**:



When we load image data, we oftentimes need to transform it to be accessible during training. For this reason, we can define a series of transformations that are going to be applied one after the other so that all the images are in the correct formatting to be used to train our CNN. Second, we **transform the full and partial masks**:



Converting mask into numpy array object.

Normalizing values into the range [0,1].

Transforming the mask into a PyTorch int tensor object.

Converting the classes to one-hot vectors.

Extracting the first index of the mask.

Now we are ready to load our data. For that, we use two PyTorch objects, the **DataSet** and the **DataLoader**. For the DataSet, we have a pre-defined class called **SliceDataset** (available in the folder **utils**, inside the file **dataset.py** that you can open in the workspace section). We create the training dataset (train_set) and the validation dataset (val-set). The directory of the data was specified in the previous cell. For both cases, we send the image transformation (transform) and the masks transformation (mask_transform).

```
[7]: batch_size = 1

'''Creating dataset objects to handle all our data samples'''
train_set = SliceDataset('train', root_dir, transform=transform, mask_transform=mask_transform, augment=True, equalize=Fals
val_set = SliceDataset('val', root_dir, transform=transform, mask_transform=mask_transform, equalize=False)

'''Creating dataloaders, which help us iterate over the data samples'''
train_loader = DataLoader(train_set, batch_size=batch_size, num_workers=4, shuffle=True)
val_loader = DataLoader(val_set, batch_size=1, num_workers=4, shuffle=False)
```

Last executed at 2022-06-08 16:05:06 in 6ms

```
>> Created train dataset with 10 images...
>> Created val dataset with 10 images...
```

Let's take a look into the **SliceDataset** class inside the dataset.py file. The __init__ method of the class initializes the object and assigns different attributes which are received when we instantiate the dataset, for example, root_dir, mask_transform, or equalize. These attributes are available inside other methods. The method __len__ is used to simply return the size of the dataset (i.e. how many images there are).

```
class SliceDataset(Dataset):
    def __init__(self, subset, root_dir, transform=None,
                 mask_transform=None, augment=False, equalize=False):
        self.root_dir: str = root_dir
        self.transform: Callable = transform
        self.mask_transform: Callable = mask_transform
        self.augmentation: bool = augment
        self.equalize: bool = equalize

        self.files = make_dataset(root_dir, subset)

        print(f">> Created {subset} dataset with {len(self)} images...")

    def __len__(self):
        return len(self.files)
```

The method called __getitem__ is in charge of reading, pre-processing, and returning each image in the dataset. In this case, for any index in the range [0, size – 1], we are going to obtain the image (img), the full mask (mask), the partial mask (weak_mask), the path of the image (img_path), the number of labels per class (true_size), and the size bounds we specify (bounds).

```python
def __getitem__(self, index) -> Dict[str, Union[Tensor, int, str]]:
    img_path, gt_path, weak_path = self.files[index]

    img = Image.open(img_path)
    mask = Image.open(gt_path)
    weak_mask = Image.open(weak_path)

    if self.equalize:
        img = ImageOps.equalize(img)

    if self.transform:
        img = self.transform(img)
        mask = self.mask_transform(mask)
        weak_mask = self.mask_transform(weak_mask)

    _, W, H = img.shape
    assert mask.shape == weak_mask.shape == (2, W, H)

    # Circle: 8011
    true_size = einsum("kwh->k", mask)
    bounds = einsum("k,b->kb", true_size, torch.tensor([0.9, 1.1], dtype=torch.float32))
    assert bounds.shape == (2, 2)  # binary, upper and lower bounds

    return {"img": img,
            "full_mask": mask,
            "weak_mask": weak_mask,
            "path": str(img_path),
            "true_size": true_size,
            "bounds": bounds}
```

Reading the image files.

Transforming the image and masks.

Making sure that both masks have the shape (num_classes, width, height)

Defining the size bounds for the two classes. The variable true_size contains the sum of labels of each class. The operation to obtain the variable bounds will define the lower and upper size bounds as 90% and 110% of the size of the region in the full mask.

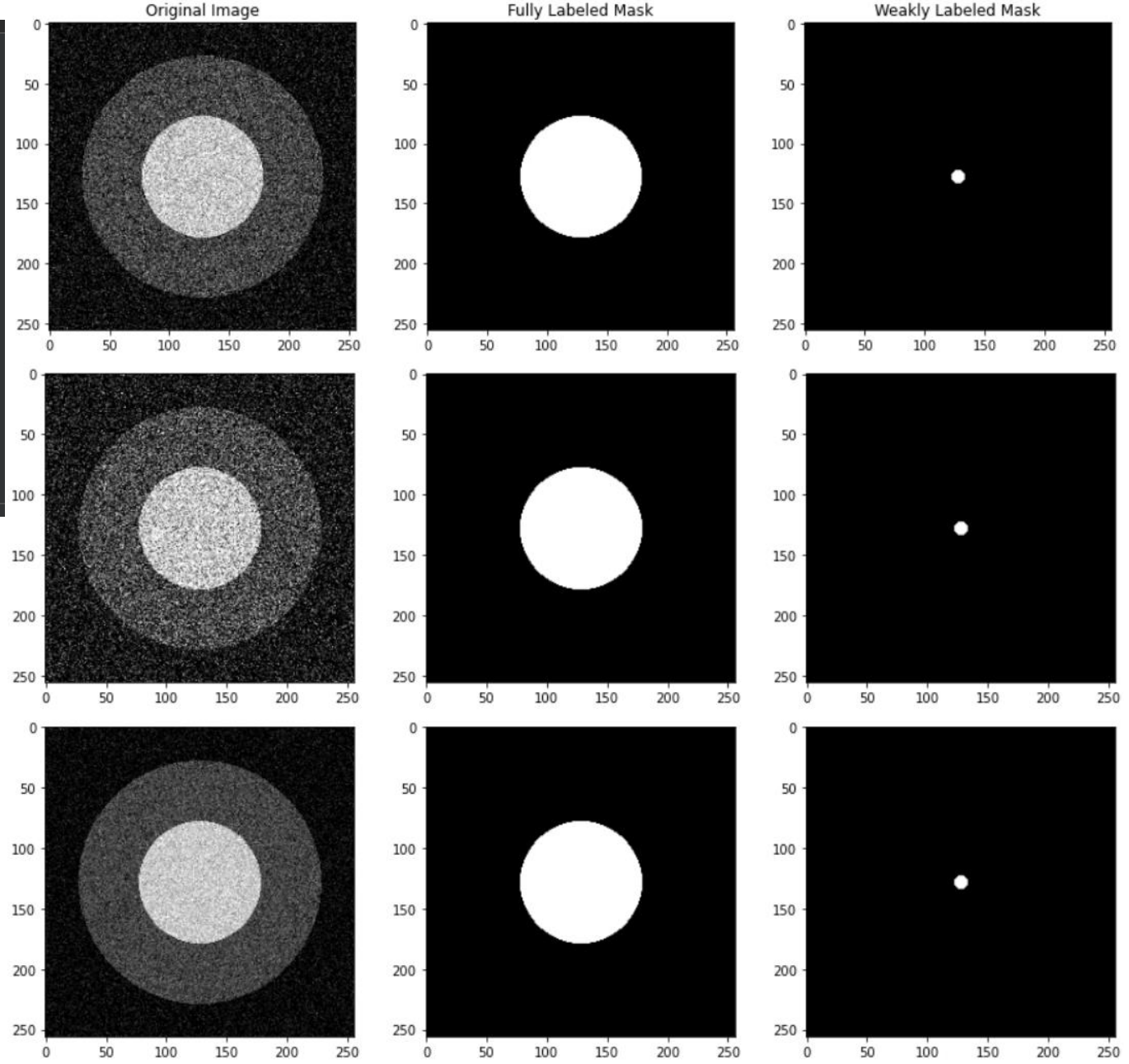
Returning the variables.

Once the dataset objects are created, we can create the dataloaders. A dataloader is simply an iterator that will help us train the CNN over the images. Notice that for the training data, shuffling is enabled, whereas for validation it is disabled. The batch size (batch_size) is 1 in this case, which means that for each iteration, we will have only one image for the forward pass in the network, and hence calculating the loss function.

```
[7]: batch_size = 1

     '''Creating dataset objects to handle all our data samples'''
     train_set = SliceDataset('train', root_dir, transform=transform, mask_transform=mask_transform, augment=True, equalize=Fal
     val_set = SliceDataset('val', root_dir, transform=transform, mask_transform=mask_transform, equalize=False)

     '''Creating dataloaders, which help us iterate over the data samples'''
     train_loader = DataLoader(train_set, batch_size=batch_size, num_workers=4, shuffle=True)
     val_loader = DataLoader(val_set, batch_size=1, num_workers=4, shuffle=False)

     Last executed at 2022-06-08 16:05:06 in 6ms

     >> Created train dataset with 10 images...
     >> Created val dataset with 10 images...
```

The following cell is a tool that allow us visualize the first three samples in the training dataset.

```
[21]: '''Displaying some examples'''
      fig, axs = plt.subplots(3, 3)
      fig.set_figheight(15)
      fig.set_figwidth(15)
      for i in range(3):
          data = train_set[i]
          img = data['img'] #Image
          full_mask = data['full_mask'] #Fully annotated segmentation mask
          weak_mask = data['weak_mask'] #Weakly annotated segmentation mask
          axs[i,0].imshow(img.squeeze(), cmap='gray')
          axs[i,1].imshow(full_mask[1], cmap='gray')
          axs[i,2].imshow(weak_mask[1], cmap='gray')
      axs[0,0].set_title('Original Image')
      axs[0,1].set_title('Fully Labeled Mask')
      axs[0,2].set_title('Weakly Labeled Mask')
      plt.show()
      Last executed at 2022-06-09 16:43:58 in 909ms
```



In the first column, we can see the original images. The second column shows the fully labeled mask containing the white circle that we want to segment from the gray one, and the third column has only a small white circle that represents the weak labeled mask (only a fraction of the total number of labels for the white circle class).

In the next cell, we create a simple CNN architecture. It is made of only three convolutional layers. Note that each layer is built with a convolutional layer (nn.Conv2d), a batch normalization layer (nn.BatchNorm2d), and PReLU as activation function (nn.PReLU).

[27]:
```
import torch.nn as nn
'''Defining a layer that contains
        ________      ____________      _______
       |        |    |            |    |       |
  -->  | conv   | --> | BatchNorm | --> | PRelU | -->
       |________|    |____________|    |_______|                '''
def convBatch(nin, nout, kernel_size=3, stride=1, padding=1, bias=False, layer=nn.Conv2d, dilation=1):
    return nn.Sequential(
        layer(nin, nout, kernel_size=kernel_size, stride=stride, padding=padding, bias=bias, dilation=dilation),
        nn.BatchNorm2d(nout),
        nn.PReLU()
    )

'''Creating a shallow Convolutional Neural Network'''
class ShallowCNN(nn.Module):
    def __init__(self, nin, nG, nout):
        super(ShallowCNN, self).__init__()
        self.conv0 = convBatch(nin, nG * 4) #First Layer
        self.conv1 = convBatch(nG * 4, nG * 4) #Second Layer
        self.conv2 = convBatch(nG * 4, nout) #Third Layer

    def forward(self, input):
        x0 = self.conv0(input)
        x1 = self.conv1(x0)
        x2 = self.conv2(x1)

        return x2

initial_kernels = 4
num_classes = 2
net = ShallowCNN(1, initial_kernels, num_classes)
net.to(device)
```

The **forward** method performs the forward pass.

Creating a network from this class.

Moving the network to the GPU.

In the following cell, we create the functions that will compute our losses; the full crossentropy loss (CrossEntropy), the partial cross-entropy loss (PartialCrossEntropy), and the size loss (NaiveSizeLoss).

[28]:
```
ce_loss = CrossEntropy(idk=[0,1]) #To supervise both background and foreground
partial_ce = PartialCrossEntropy() #To supervise only forefround
sizeloss = NaiveSizeLoss()
```

```
Initialized CrossEntropy with {}
Initialized PartialCrossEntropy with {}
Initialized NaiveSizeLoss with {}
```

The source code of the three losses can be found in the folder **utils**, inside the file named **losses.py**.

The cross-entropy loss is also defined as a class. The __init__ method initializes the metric by receiving the indices corresponding to the different classes to segment (two classes in this case). The method __call__ simply performs the calculation. The inputs of the loss are the softmax predictions and targets.

```python
class CrossEntropy():
    def __init__(self, idk, **kwargs):
        # Self.idk is used to filter out some classes of the target mask. Use fancy indexing
        self.idk = idk
        print(f"Initialized {self.__class__.__name__} with {kwargs}")

    def __call__(self, pred_softmax, weak_target):
        assert pred_softmax.shape == weak_target.shape
        assert simplex(pred_softmax)
        assert sset(weak_target, [0, 1])

        log_p = (pred_softmax[:, self.idk, ...] + 1e-10).log()
        mask = weak_target[:, self.idk, ...].float()

        loss = - einsum("bkwh,bkwh->", mask, log_p)
        loss /= mask.sum() + 1e-10

        return loss
```

Verification. We check if the predictions and targets have the same shape, if the softmax predictions are correct probability vectors, and if the targets correspond to the available classes.

Computing the log of the predictions and extracting the one-hot vectors corresponding to each pixel.

We calculate the proper cross-entropy loss using the one-hot vectors and the log values of the predicted probabilities, and average it at the end.

The PartialCrossEntropy loss is built as a class that inherits its features from the CrossEntropy class. The main difference is that here we only use one index, corresponding to the class from which we have partial labels (instead of having full annotation).

```python
class PartialCrossEntropy(CrossEntropy):
    def __init__(self, **kwargs):
        super().__init__(idk=[1], **kwargs)
```

The NaiveSizeLoss computes the size loss with all the pixels, disregarding of labeling. The idea is to penalize the supervised loss (cross-entropy), based on the sizes of the predicted segmented regions. Here we use the bounds we pre-defined as the size priors for our images.

```python
class NaiveSizeLoss():
    """
    This one implement the naive quadratic penalty
    penalty = 0                 if a <= pred_size
              (a - pred_size)^2 otherwise
    """
    def __init__(self, **kwargs):
        # Self.idk is used to filter out some classes of the target mask. Use fancy indexing
        self.idk = [1]
        print(f"Initialized {self.__class__.__name__} with {kwargs}")

    def __call__(self, pred_softmax, bounds):
        assert simplex(pred_softmax)

        B, K, H, W = pred_softmax.shape
        assert bounds.shape == (B, K, 2)

        pred_size = einsum("bkwh->bk", pred_softmax)[:, self.idk]

        upper_bounds = bounds[:, self.idk, 1]
        lower_bounds = bounds[:, self.idk, 0]
        assert (upper_bounds >= 0).all() and (lower_bounds >= 0).all()

        # size < upper <==> size - upper < 0
        # lower < size <==> lower - size < 0
        loss = F.relu(pred_size - upper_bounds) ** 2 + F.relu(lower_bounds - pred_size) ** 2
        loss /= (W * H)

        return loss
```

Verification. Extracting the shape of the softmax predictions to compare them with the shape of the bounds.

Extracting upper and lower bounds.

Automatic way of applying the size constrain, using ReLU to check if the predicted region's size is larger or smaller than the lower and upper bounds.

In the next cell we configure the training settings. These include the learning rate (lr), the number of epochs of training (epochs), the optimizer to use, which is Adam in this case, and the training mode. For the latter, we have three options: **full**, **unconstrained**, and **constrained**, each corresponding to settings of fully supervised training, weakly supervised training, and weakly supervised training with the size loss constrain.

```
epochs = 150 #Number of epochs to train
lr = 0.0005 #Learning rate
optimizer = torch.optim.Adam(net.parameters(), lr=lr, betas=(0.9, 0.999)) #Optimizer
mode = 'full'
#mode = 'unconstrained'
#mode = 'constrained'
```

The first configuration to run is **full**, as the fourth line in this cell is uncommented. After that, if we want to run the others, we simply need to uncomment (to erase the # before the corresponding line) the following two lines, one by one.

We now execute our training cell. Here we iterate over the epochs, to then iterate over the batches of our dataloader.

```
for i in range(epochs):
    net.train()

    '''Tensors to save loss values'''
    log_ce = torch.zeros((len(train_loader)), device=device)
    log_sizeloss = torch.zeros((len(train_loader)), device=device)
    log_sizediff = torch.zeros((len(train_loader)), device=device)
    log_dice = torch.zeros((len(train_loader)), device=device)

    desc = f">> Training   ({i: 4d})"
    tq_iter = tqdm_(enumerate(train_loader), total=len(train_loader), desc=desc)
```

Starting iterations. The network should be passed to train mode.

Creating tensors to save metrics and compute averages at the end of each epoch.

Creating iterator for the training dataloader.

We iterate over the batches of the dataloader.

```
for j, data in tq_iter:
    img = data['img'].to(device)
    full_mask = data['full_mask'].to(device)
    weak_mask = data['weak_mask'].to(device)
    bounds = data['bounds'].to(device)

    optimizer.zero_grad(set_to_none=True)

    # Sanity tests to see we loaded and encoded the data correctly
    assert 0 <= img.min() and img.max() <= 1
    B, _, W, H = img.shape
    assert B == 1  # Since we log the values in a simple way, doesn't handle more
    assert weak_mask.shape == (B, 2, W, H)
    assert one_hot(weak_mask), one_hot(weak_mask)

    '''Forward pass'''
    logits = net(img)
    pred_softmax = F.softmax(5*logits, dim=1)
    pred_seg = probs2one_hot(pred_softmax) #Segmentation prediction

    pred_size = einsum('bkwh->bk', pred_seg)[:,1]
    log_sizediff[j] = pred_size - data['true_size'][0,1]
    log_dice[j] = dice_coef(pred_seg, full_mask)[0,1]
```

We read the images, the full mask, the weak mask, and the bounds from each batch.

Verification of image values and shape. Checking that the weak mask is composed of one-hot vectors.

Forward pass. We compute the logits of the network and transform the results into the softmax predictions. We convert the probability vectors into one-hot vectors.

We calculate the total size of the predicted segmentation region and compare it with the true size of the mask. Also compute the dice coefficient.

The calculation of the loss function will depend on the training mode we have selected at the beginning.

```python
if mode == 'full':
    '''Using standard supervision with CrossEntropy Loss'''
    ce_val = ce_loss(pred_softmax, full_mask)
    log_ce[j] = ce_val.item()

    log_sizeloss[j] = 0

    lossEpoch= ce_val
elif mode == 'unconstrained':
    ce_val = partial_ce(pred_softmax, weak_mask)
    log_ce[j] = ce_val.item()

    log_sizeloss[j] = 0

    lossEpoch = ce_val
elif mode == 'constrained':
    '''Using sizeloss along with CrossEntropy with weak supervision'''
    ce_val = partial_ce(pred_softmax, weak_mask)
    log_ce[j] = ce_val.item()

    sizeLoss_val = sizeloss(pred_softmax, bounds)
    log_sizeloss[j] = sizeLoss_val.item()

    lossEpoch = ce_val + sizeLoss_val / 100
```

For the full training mode, we use the standard cross-entropy loss using the full mask.

For the unconstrained mode, we use the partial cross-entropy loss, only having access to the partial mask.

For the constrained mode, we use the partial cross-entropy loss with the partial mask, as well as size loss to constrain the learning of the CNN.

At the end of this cell, we perform the backward pass with the corresponding loss function we had. We print metrics such as dice coefficient, size difference, cross-entropy loss, and size loss. Each five epochs, we also evaluate on the validation set and

Backward pass

Update the network parameters.

```
        '''Backward pass'''
        lossEpoch.backward()
        optimizer.step()
        tq_iter.set_postfix({"DSC": f"{log_dice[:j+1].mean():05.3f}",
                             "SizeDiff": f"{log_sizediff[:j+1].mean():07.1f}",
                             "LossCE": f"{log_ce[:j+1].mean():5.2e}",
                             **({"LossSize": f"{log_sizeloss[:j+1].mean():5.2e}"} if mode == 'constrained' else {})})
        tq_iter.update(1)
    tq_iter.close()
    if (i % 5) == 0:
        saveImages(net, val_loader, 1, i, 'TOY', mode, device) #Evaluate on the evaluation set and save segmented images
```

Printing metrics at each epoch and evaluating each five epochs.

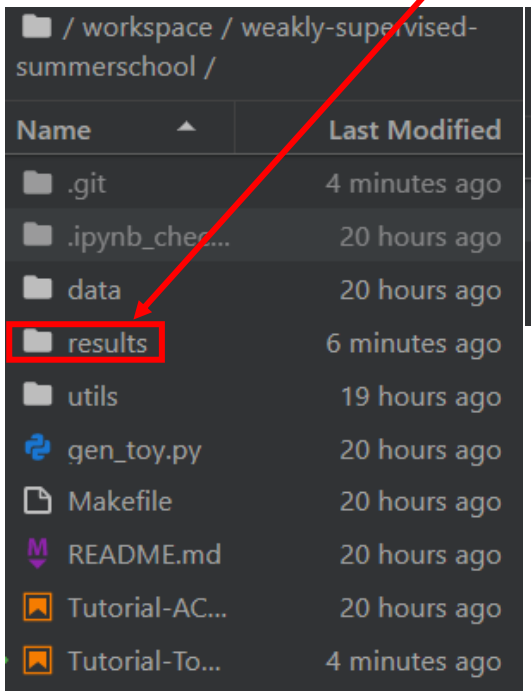
We will run the training three times, one per configuration, and analyze the results with each of them.

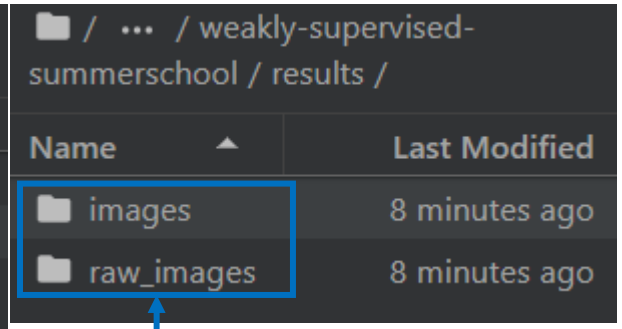
When running the **full** configuration, we observe a natural decrease in the size difference between the regions, as well as in the cross-entropy loss. Also, the dice score increases, showing that a better segmentation performance has been achieved.

```
>> Training   (   0): 100%|                                                  | 10/10 [25.80it/s, DSC=0.425, SizeDiff=20514.8, LossCE=7.99e-01]
```

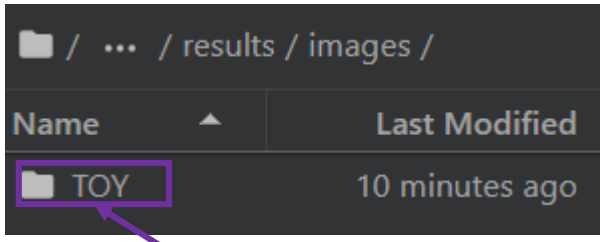
```
>> Training   ( 149): 100%|                                                  | 10/10 [27.35it/s, DSC=0.995, SizeDiff=00007.9, LossCE=7.60e-03]
```

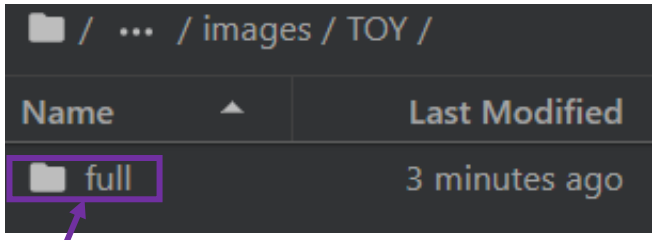
After running this cell, a new folder called **results** will appear in the workspace section.





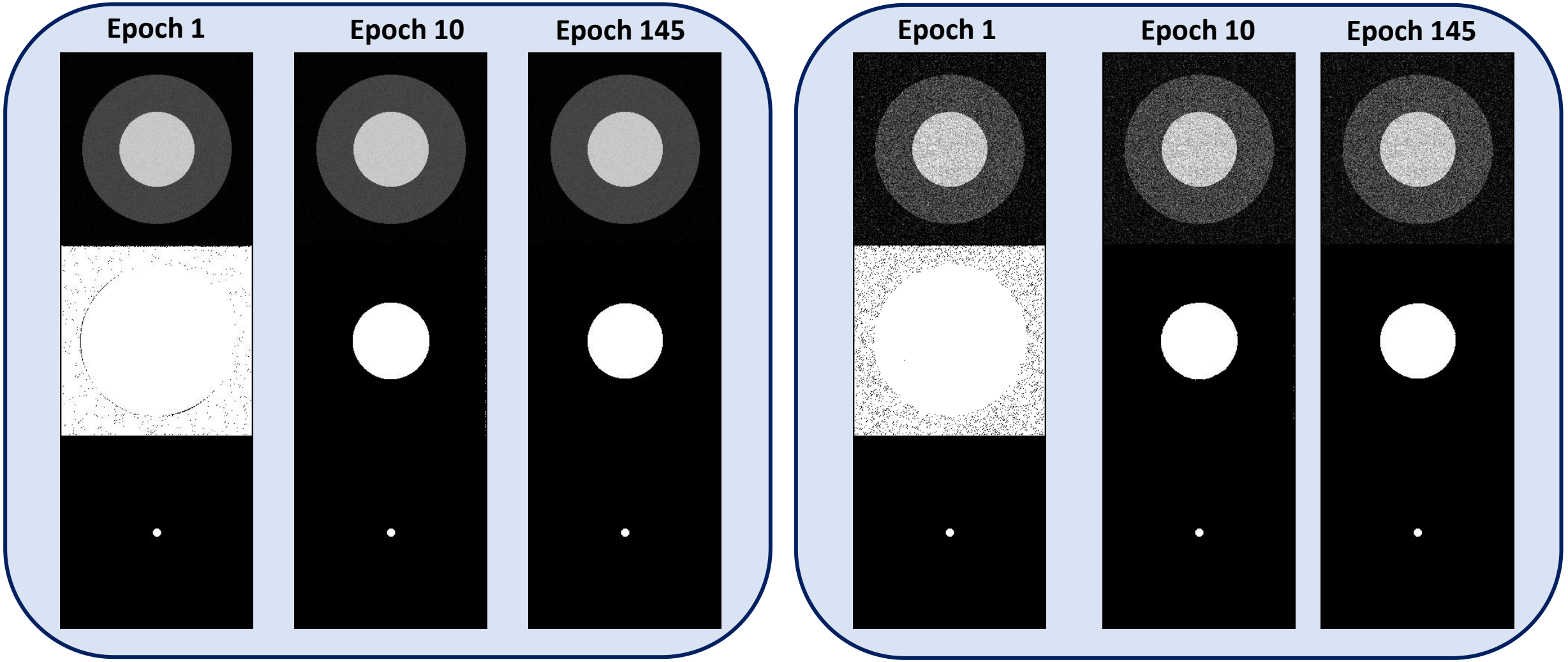
In the inside, we see two new subfolders, one called **images** and the other one called **raw_images**.





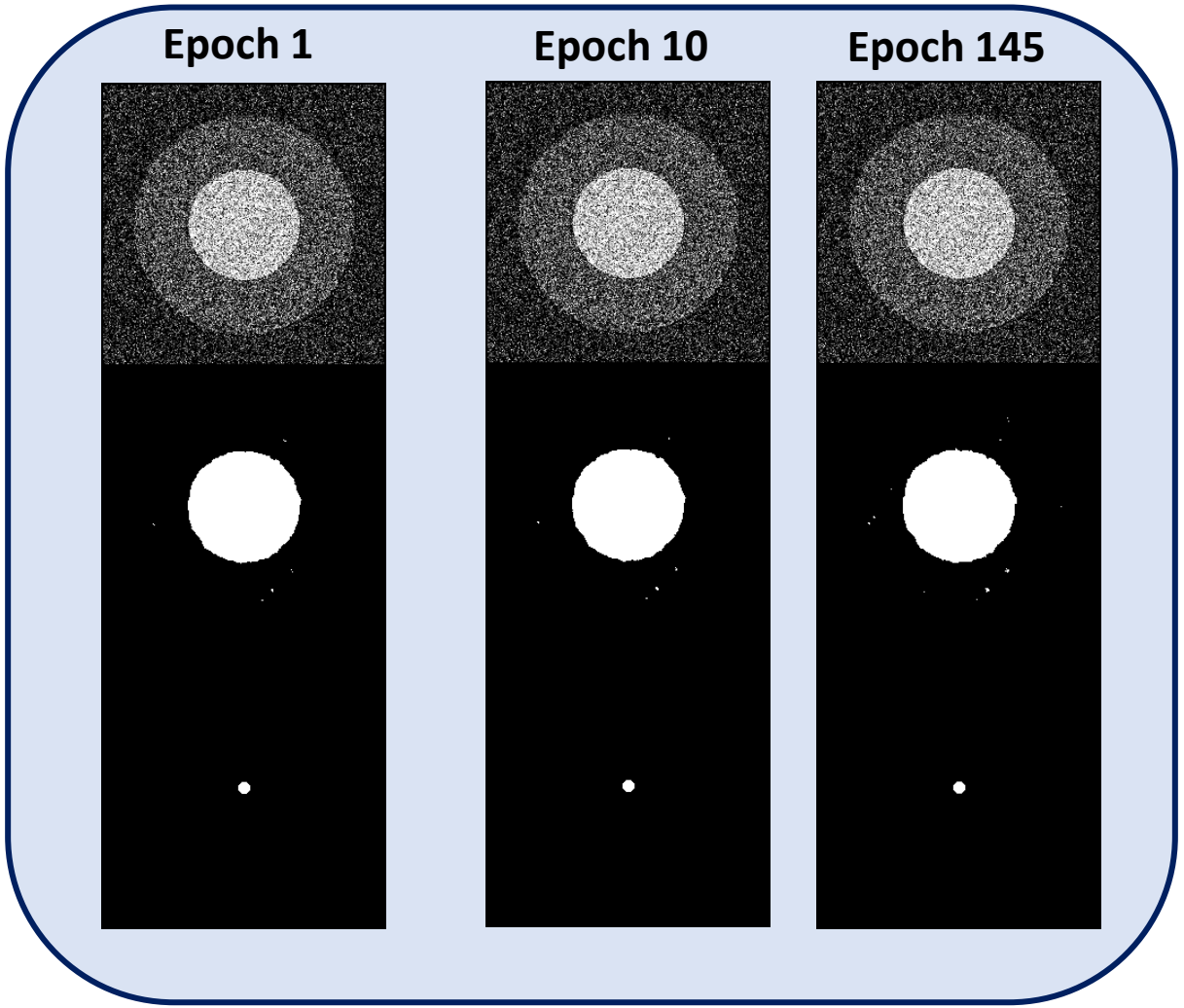
Inside the **images**, a folder called **TOY** will be available. Furthermore, a subfolder corresponding to each configuration will be inside this latter directory. In this case, we will see it with the name **full**, which is the configuration we have just run.

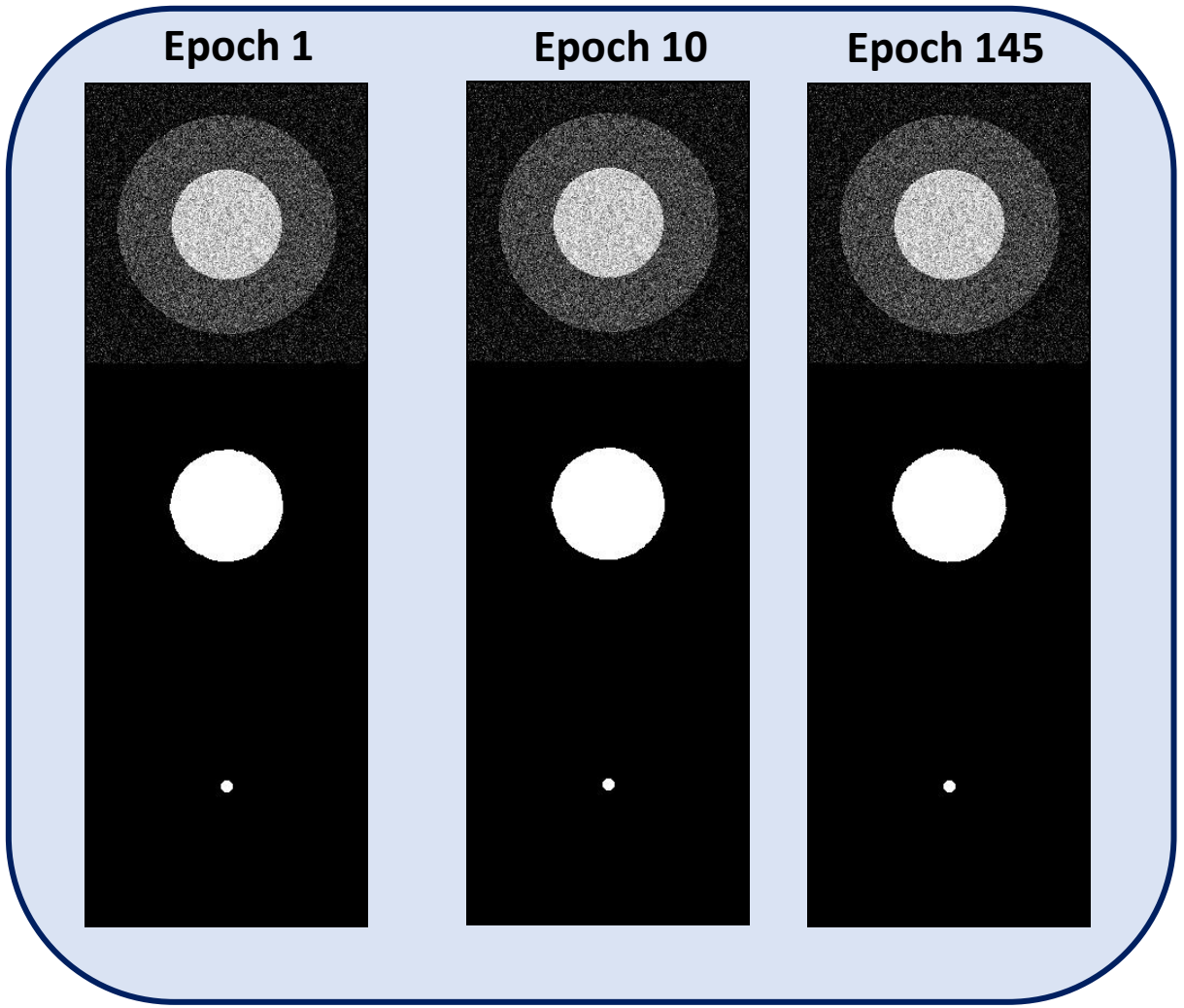
Inside the last subfolder, we will see a list of images. Each image has the following nomenclature: imageNumber_Ep_00epoch.png. For example, the first image in the validation set, at the first epoch, will appear as 0_Ep_0000.png. Below we can see the first and sixth images, and the evolution of their segmentation using full annotation. As expected, the network is capable of outputting the correct segmentation mask for the white circle, even with the noise present in the images.



We now uncomment the line mode = ‘unconstrained’ and train the model again. This time, we will just use the weak masks and the partial cross-entropy for the training of the model. At the end, we expect a segmentation performance not so good as with full supervision. The results are acceptable at the end, but the boundaries of the circle are less well defined.

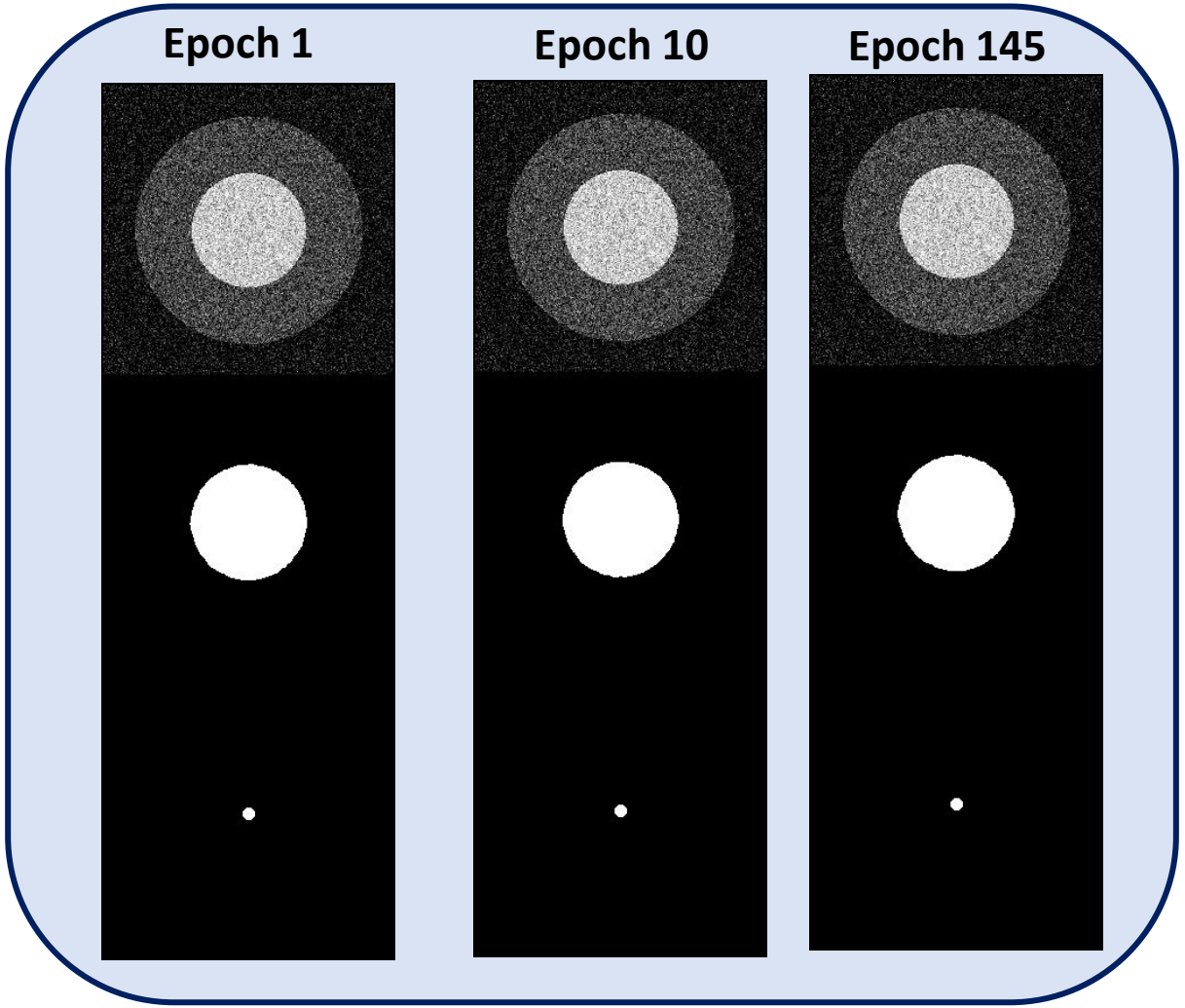
```
>> Training   (   0): 100%| | 10/10 [25.97it/s, DSC=0.993, SizeDiff=00086.9, LossCE=2.33e-04]
>> Training   ( 149): 100%| | 10/10 [27.52it/s, DSC=0.976, SizeDiff=00394.1, LossCE=1.37e-07]
```

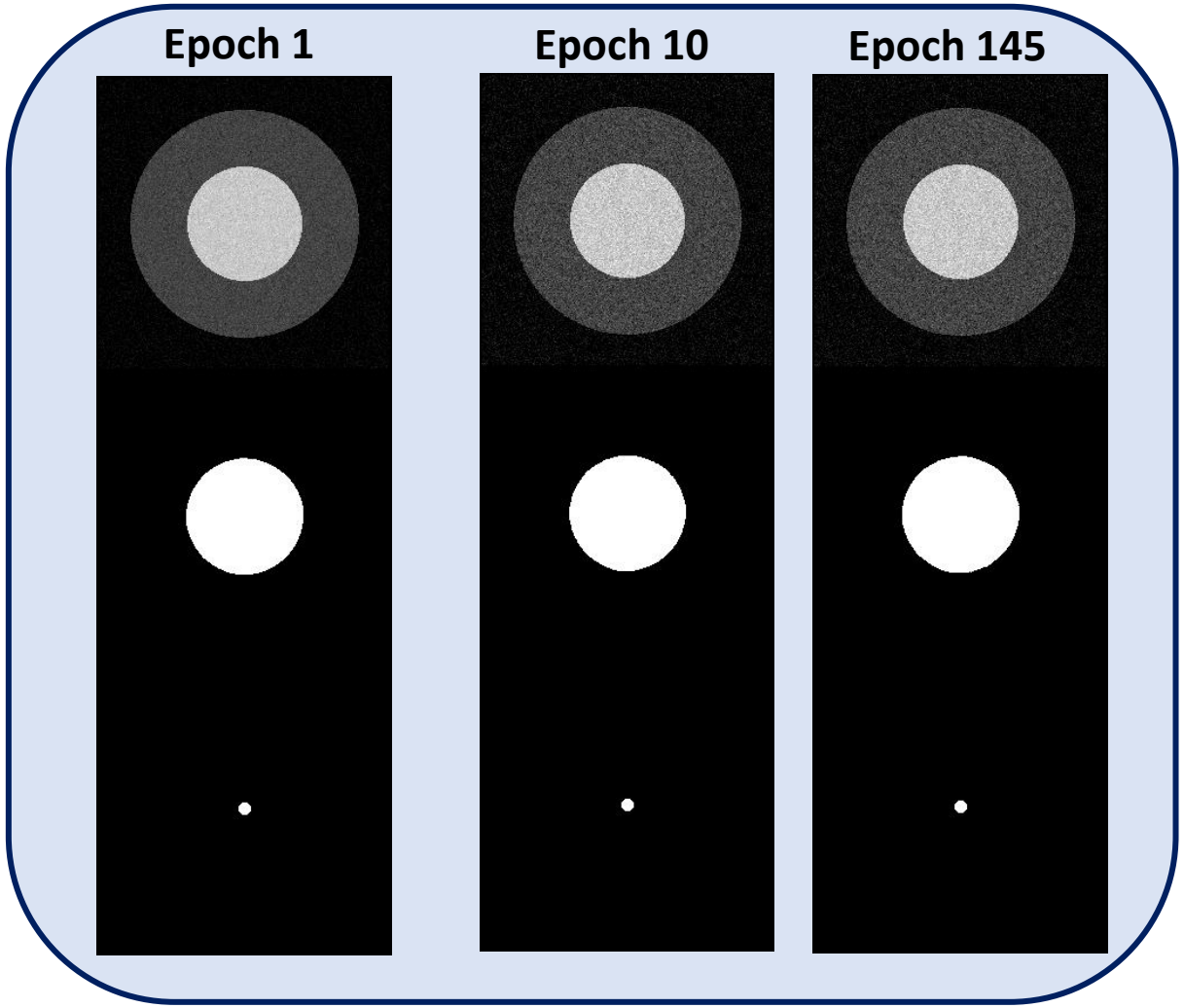




Finally, for the **constrained** approach, we use the partial cross-entropy loss along with the size loss. Once again, we have just access to the weak masks. The results are pretty competitive even at the first epochs of training.

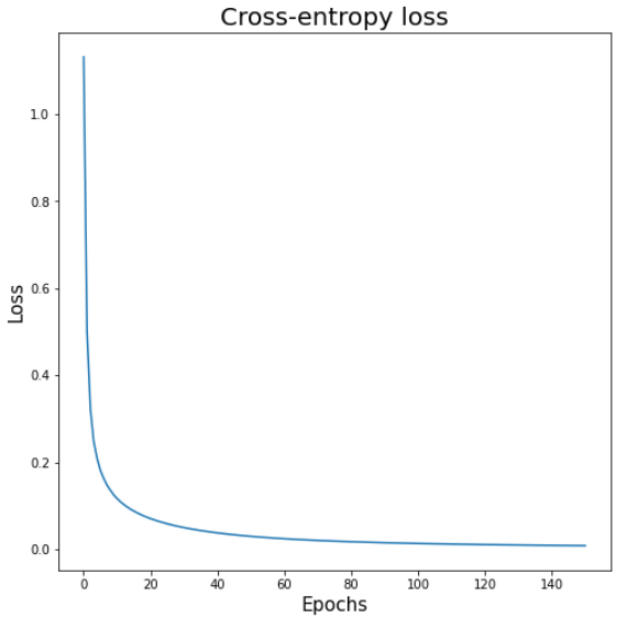
```
>> Training   (   0): 100%|| 10/10 [26.36it/s, DSC=0.984, SizeDiff=00253.0, LossCE=5.15e-07, LossSize=1.60e-01]
>> Training   ( 149): 100%|| 10/10 [26.90it/s, DSC=0.988, SizeDiff=00163.8, LossCE=2.96e-07, LossSize=0.00e+00]
```



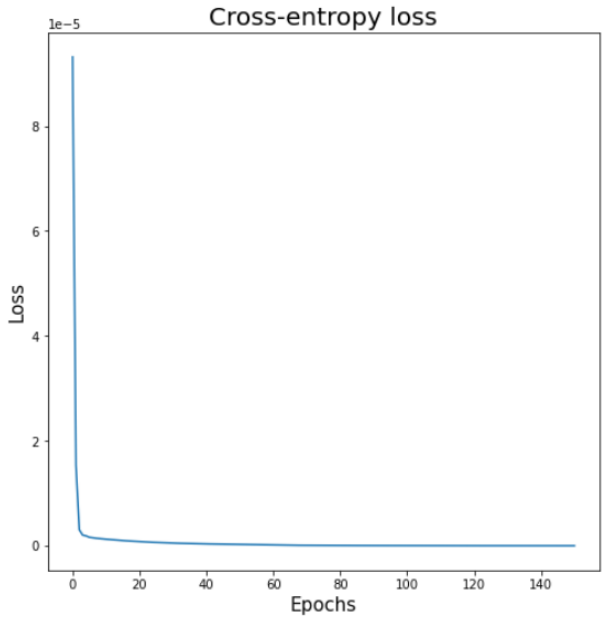


We can plot the cross-entropy loss in the three scenarios to observe the effects of the training of the model using full labels, weak labels without constraints, and constraining the loss. The behavior of the cross-entropy loss when using size loss as constraint changes with respect to the full supervised loss and the partial cross-entropy loss. We now see a sudden increase in the loss values followed by a smooth decrease.

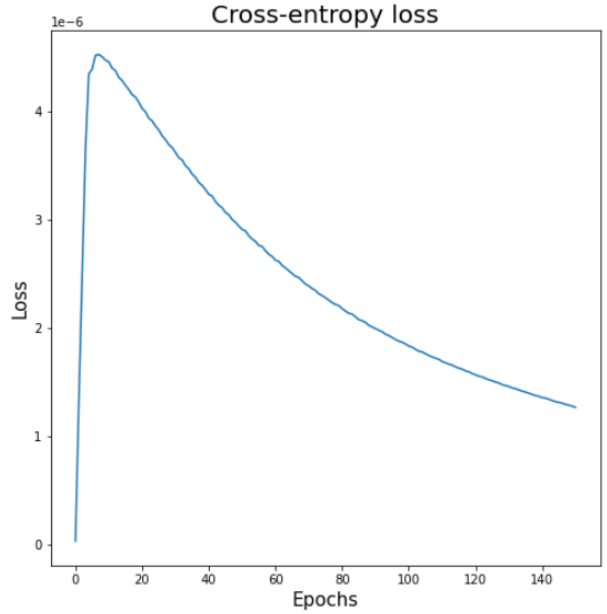
```
[15]: '''Displaying some examples'''
      fig = plt.figure(0)
      fig.set_figheight(8)
      fig.set_figwidth(8)

      plt.plot(np.arange(epochs), losses)

      plt.title('Cross-entropy loss', fontsize=20)
      plt.xlabel('Epochs', fontsize=15)
      plt.ylabel('Loss', fontsize=15)
      plt.show()
      Last executed at 2022-06-14 18:32:17 in 118ms
```



Fully supervision



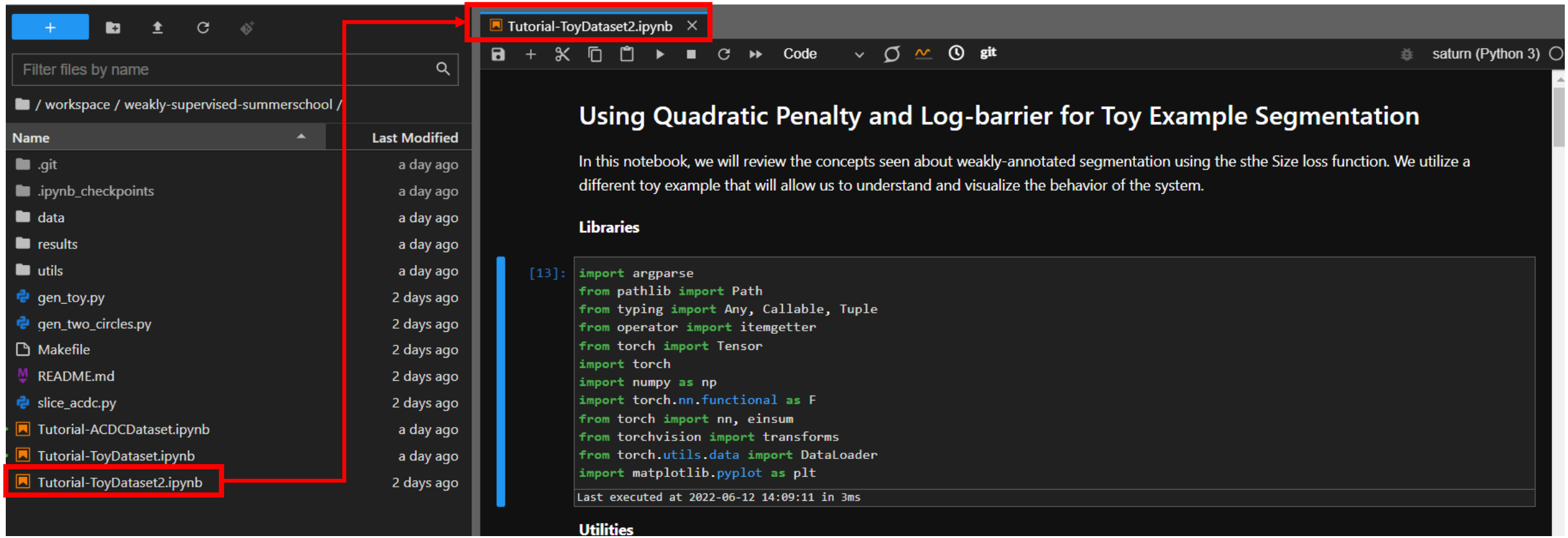
Weak supervision



Weak supervision + constraint

Weakly supervised segmentation: Log-barrier extensions

For demonstrating how to build the Log-barrier extension, we will open the Jupyter notebook called **Tutorial-ToyDataset2.ipynb**. The code in this notebook is very similar to the previous one, therefore we will focus on the parts that present differences.



We can notice now that the directory from which we are going to load the data has changed for the toy dataset 2. However, the same transformations still apply.

Loading the Toy dataset

[4]:
```
root_dir = 'data/TOY2/'

'''Specifying the transforms for the data during training'''
transform = transforms.Compose([
                lambda img: img.convert('L'),
                lambda img: np.array(img)[np.newaxis, ...],
                lambda nd: nd /255,
                lambda nd: torch.tensor(nd, dtype=torch.float32)])

mask_transform = transforms.Compose([
                    lambda img: np.array(img)[...],
                    lambda nd: nd / 255,
                    lambda nd: torch.tensor(nd, dtype=torch.int64)[None, ...],
                    lambda t: class2one_hot(t, K=2),
                    itemgetter(0)])
```

We change the directory to load the toy dataset 2.

Last executed at 2022-06-12 14:04:38 in 5ms

For this experiment, we now have two modes. The **quadratic** mode uses a quadratic penalty as loss function, which corresponds to size loss we learned about in the previous part. The **logbarrier** mode implies that we utilize the Log-barrier extension instead of the size loss.

Setting training configurations

```
•[64]: epochs = 30 #Number of epochs to train
       lr = 0.0005 #Learning rate
       optimizer = torch.optim.Adam(net.parameters(), lr=lr, betas=(0.9, 0.999)) #Optimizer
       mode = 'quadratic'
       mode = 'logbarrier'
```

New training modes.

```
Last executed at 2022-06-14 16:58:23 in 3ms
```

The quadratic penalty and the log-barrier extension are defined as classes. To create an instance from them, we pass the index of the class we want to ignore (background) and an auxiliary function that is used to compute the metric we want to constrain; **soft_size** for the size of the region, and **soft_centroid** for the centroid of the region. Both functions are defined inside the file **utils** in the folder of the same name.

```
[65]: Loss_Size: Callable[[Tensor, Tensor], Tensor]
      Loss_Centroid: Callable[[Tensor, Tensor], Tensor]
      if mode == "quadratic":
          Loss_Size = ParametrableQuadraticPenalty(idk = [0], function=soft_size)
          Loss_Centroid = ParametrableQuadraticPenalty(idk = [0], function=soft_centroid)
      elif mode == "logbarrier":
          Loss_Size = ParametrableLogBarrier(idk = [0], function=soft_size)
          Loss_Centroid = ParametrableLogBarrier(idk = [0], function=soft_centroid)
```

Quadratic penalties constraining the size of the target region and its centroid.

Log-barrier extension penalties constraining the size of the target region and its centroid.

```
Last executed at 2022-06-14 16:58:25 in 5ms
Initialized ParametrableLogBarrier with {'idk': [0], 'function': <function soft_size at 0x7f507d6cddc0>}
Initialized ParametrableLogBarrier with {'idk': [0], 'function': <function soft_centroid at 0x7f507d6cde50>}
```

The class ParametrableLogBarrier is used as the log-barrier extension penalty function. We initialize the function with the indices of the classes we want to filter, the function to compute the metric to be constrained, and defining the hyperparameter t.

```python
# Quadratic penalty
class ParametrableLogBarrier():
    """
    This one implement the naive quadratic penalty
    penalty = 0                 if a <= pred_size
              (a - pred_size)^2 otherwise
    """
    def __init__(self, **kwargs):
        # Self.idk is used to filter out some classes of the target mask. Use fancy indexing
        self.idk = kwargs['idk']
        self.function: Callable[[Tensor], Tensor] = kwargs["function"]
        self.t: float = 1
        print(f"Initialized {self.__class__.__name__} with {kwargs}")

    def __barrier__(self, z: Tensor) -> Tensor:
        assert z.shape == ()

        if z <= - 1 / self.t**2:
            res = - torch.log(-z) / self.t
        else:
            res = self.t * z + -np.log(1 / (self.t**2)) / self.t + 1 / self.t

        assert res.requires_grad == z.requires_grad
        # # print(res)

        return res
```

Initializing the log-barrier extension class. A function is needed to compute the metric used for the constraint.

The __barrier__ method calculates the function by parts defined previously for this penalty function.

The __call__ method puts everything together to calculate the actual loss function.

```python
def __call__(self, pred_softmax, bounds):
    assert simplex(pred_softmax)

    B, K, H, W = pred_softmax.shape
    _, _, M, _ = bounds.shape
    assert bounds.shape == (B, K, M, 2)

    pred_fn = self.function(pred_softmax)[:, self.idk].view(B, len(self.idk), M)

    upper_bounds = bounds[:, self.idk, :, 1]
    lower_bounds = bounds[:, self.idk, :, 0]

    assert pred_fn.shape == upper_bounds.shape == lower_bounds.shape

    # pred < upper <==> pred - upper < 0
    # lower < pred <==> lower - pred < 0
    upper_part: Tensor = reduce(add, (self.__barrier__(z) for z in (pred_fn - upper_bounds).flatten()))
    lower_part: Tensor = reduce(add, (self.__barrier__(z) for z in (lower_bounds - pred_fn).flatten()))

    loss: Tensor = upper_part + lower_part
    loss /= (W * H)

    return loss
```

Verification of softmax predictions shape and values, as well as of the bounds.

Computing the metric to be constrained.

Extracting the upper and lower bounds and checking shape compatibility with the metric.

Comparison between the prediction metric and the lower and upper bounds using the log-barrier extension.

The training phase now changes a little bit for these new training modes. However, the same structure and tools used for training in the previous section apply here.

```python
losses = []
for i in range(epochs):
    net.train()

    log_dice = torch.zeros((len(train_loader)), device=device)
    loss = torch.zeros((len(train_loader)), device=device)

    desc = f">> Training   ({i: 4d})"
    tq_iter = tqdm_(enumerate(train_loader), total=len(train_loader), desc=desc)

    for j, data in tq_iter:
        img = data["img"].to(device)
        full_mask = data["full_mask"].to(device)

        # Sanity tests to see we loaded and encoded the data correctly
        assert 0 <= img.min() and img.max() <= 1
        B, _, W, H = img.shape
        _, K, _, _ = full_mask.shape
        assert B == 1  # Since we log the values in a simple way, doesn't handle more

        true_size: Tensor = soft_size(full_mask)[..., None]  # Add an extra axis
        assert true_size.shape == (B, K, 1)  # last one is dimensionality of the value computed (size)
        true_centroid: Tensor = soft_centroid(full_mask)
        assert true_centroid.shape == (B, K, 2)  # Dimensionality is two for the centroid (two axis)

        bounds_size = einsum("bkm,u->bkmu", true_size, torch.tensor([0.9, 1.1],
                                                                    dtype=torch.float32,
                                                                    device=true_size.device))
        bounds_centroid = einsum("bkm,u->bkmu", true_centroid, torch.tensor([0.9, 1.1],
                                                                            dtype=torch.float32,
                                                                            device=true_size.device))
```

Large verification portion of the code. Here we make sure the shape of the data is compatible with the mask. We also make sure that the calculation of the size and centroid of the target regions have the shape we need to compare them with the bounds.

Computing the lower and upper bounds of the size and centroid to be used in the calculation of the penalty functions.

```
optimizer.zero_grad()

#Forward pass
logits = net(img)
pred_softmax = F.softmax(5 * logits, dim=1)
pred_seg = probs2one_hot(pred_softmax)
```

Forward pass and computation of the softmax predictions.

```
log_dice[j] = dice_coef(pred_seg, full_mask)[0, 1]  # 1st item, 2nd class
```

Calculating the DICE score for segmentation.

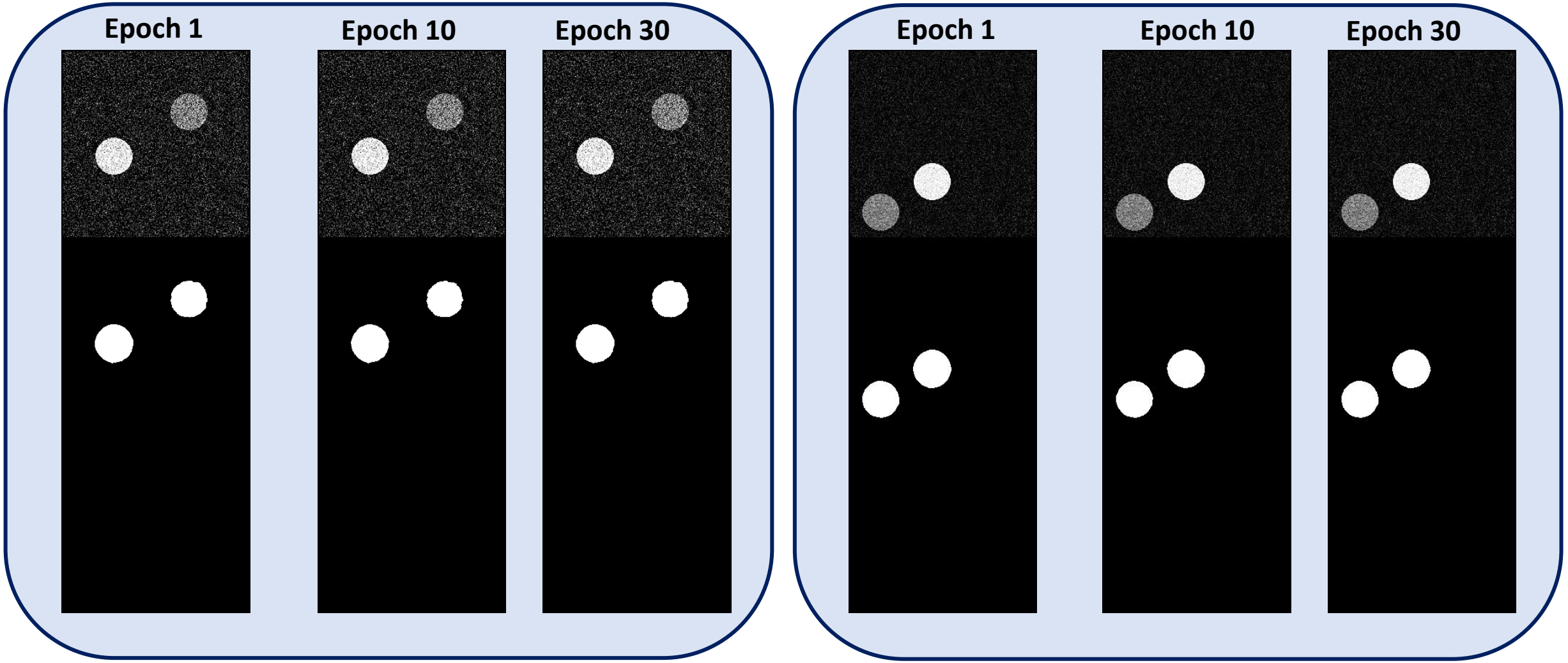
```
combined_loss = Loss_Size(pred_softmax, bounds_size) + Loss_Centroid(pred_softmax, bounds_centroid)
loss[j] = combined_loss.item()

combined_loss.backward()
optimizer.step()

tq_iter.set_postfix({"DSC": f"{log_dice[:j+1].mean():05.3f}"})
tq_iter.update(1)
```

The loss function combines the penalty for the centroid of the target region and the penalty of the size of the size. We also do the backward pass and update the network parameters.

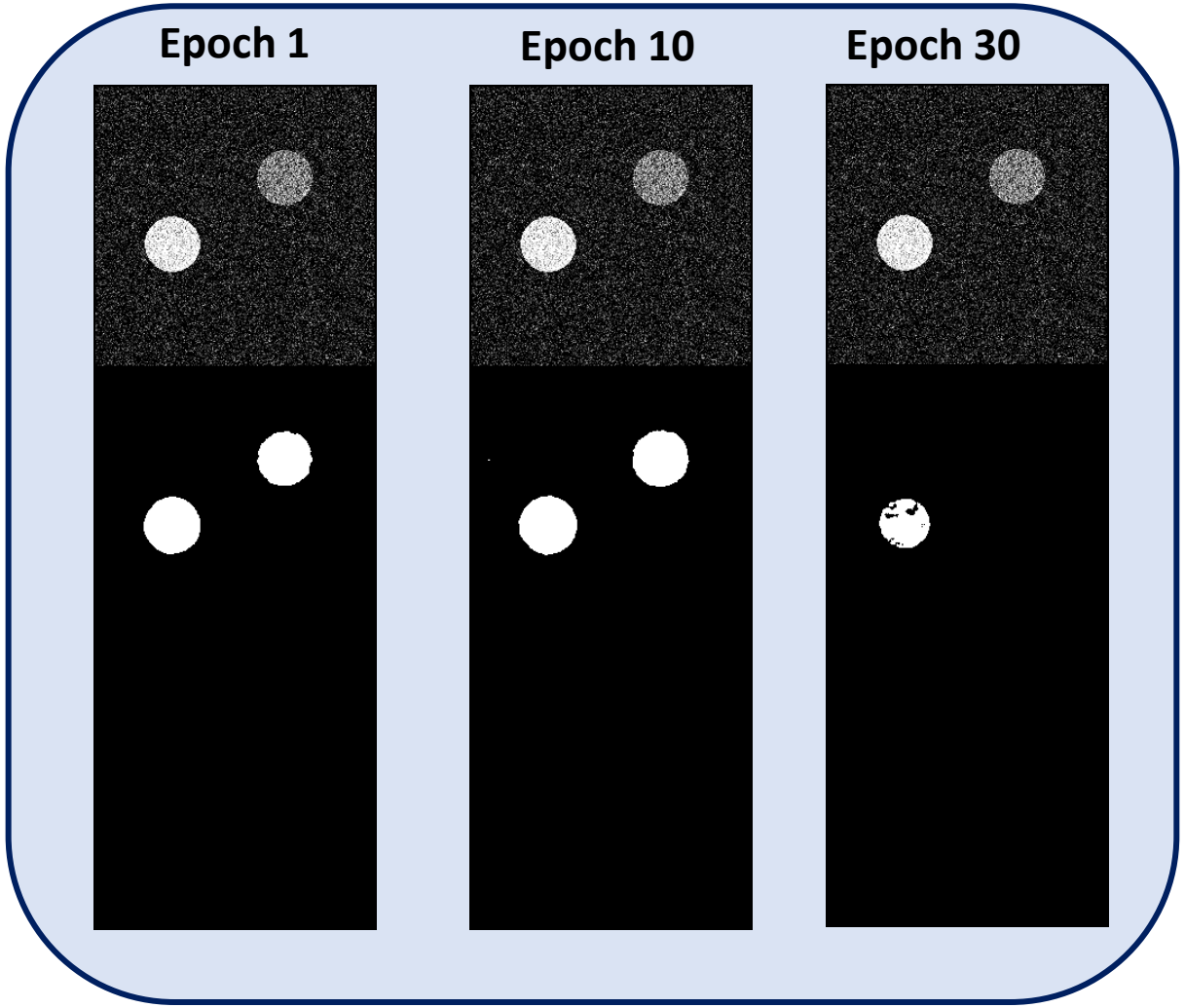
```
if mode == 'logbarrier':
    Loss_Size.t *= 1.1
    Loss_Centroid.t *= 1.1
```

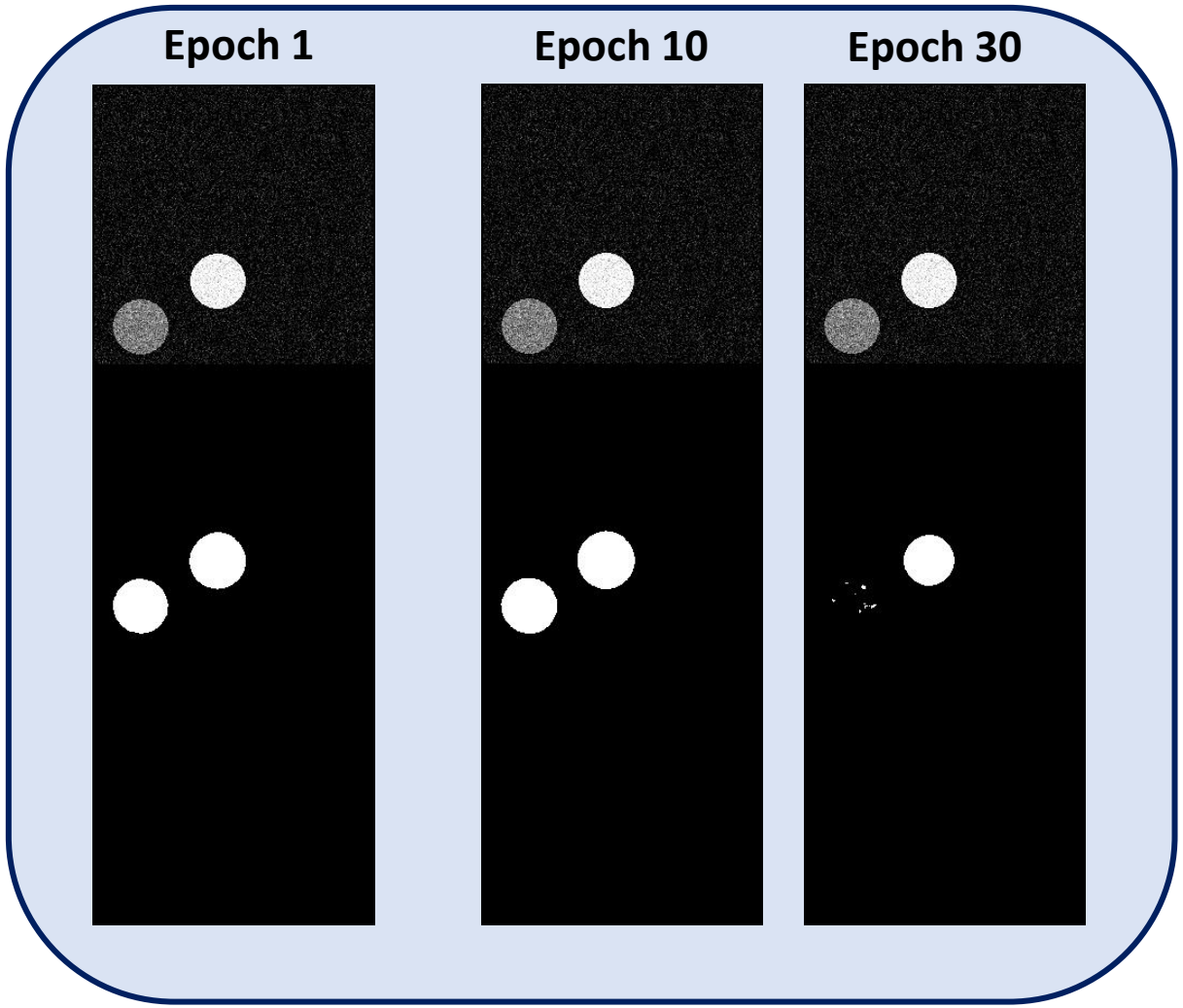
When using the log-barrier extension, we make sure of increasing the hyperparameter t a small percent of its current size. This has the effect of forcing the predictions to preserve their good values even at late stages of the training.

We first execute the training with the **quadratic** mode. It is important to notice that the we do not use any supervision when we calculate the penalties. With a more complex dataset, a weakly supervision would be also desirable, however, we can see that with a very simple dataset, labels are mostly not required. Our goal is to segment the white circle, and what we achieve is to segment both just by using the size and centroid priors.



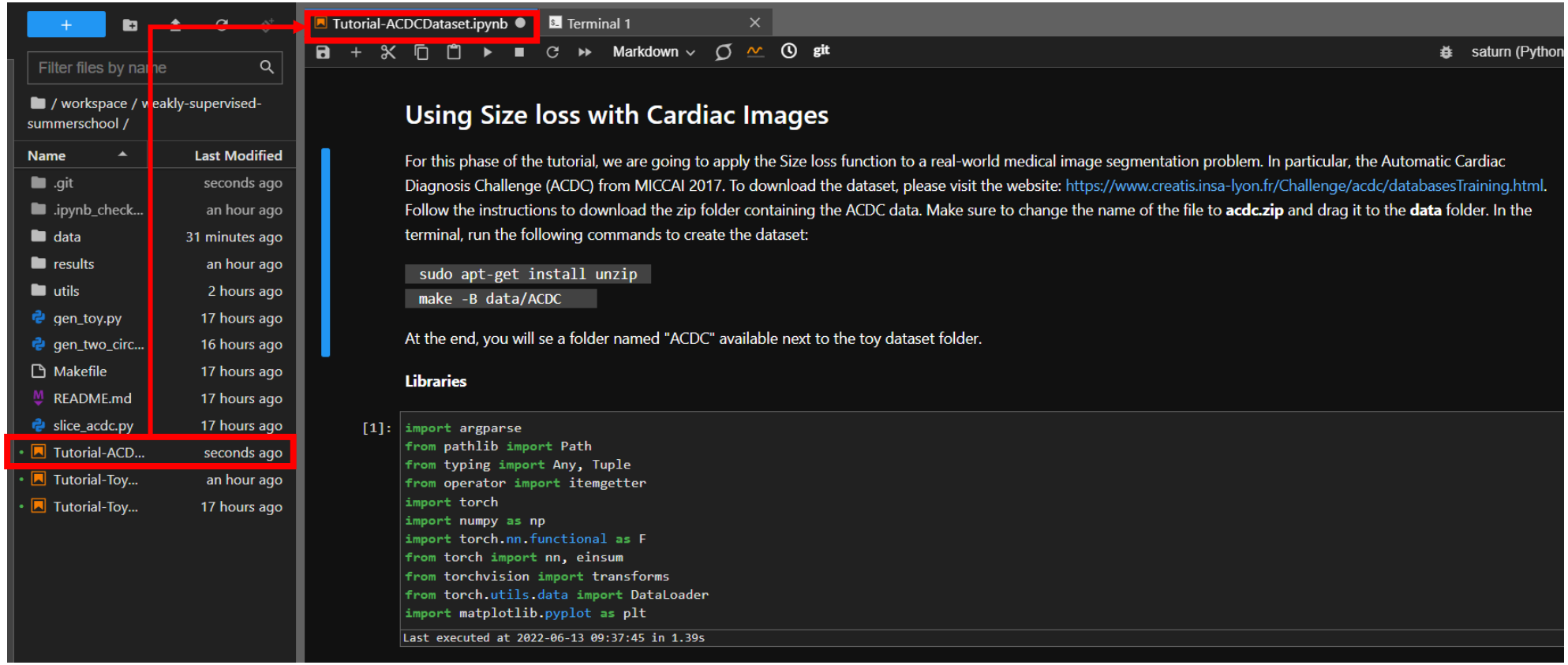
We can also obtain very good results with the **logbarrier** mode. Particularly, this approach is very powerful, as the log-barrier extensions allow us to utilize the log-barrier method for training CNNs without requiring an initial solution that is feasible with respect to the constraints. Once again, without supervision, segmentation results are impressive with this toy dataset. Here we achieve to segment only the white circle.





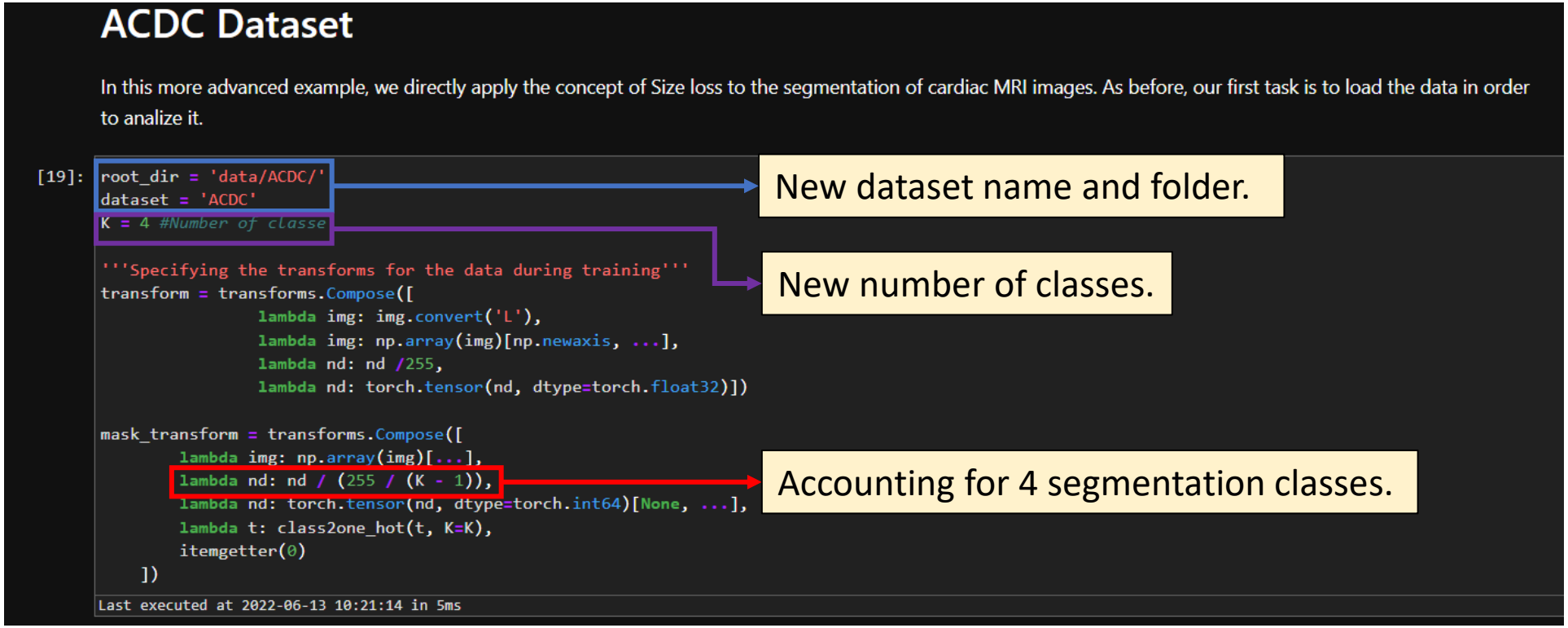
Weakly supervised segmentation: ACDC dataset

Back in the **weakly-supervised-summerschool** folder (root folder), we will open the Jupyter notebook called **Tutorial-ACDCDataset.ipynb**. This notebook contains the code we need to run the weakly supervised segmentation method of the cardiac images we downloaded and prepared at the beginning.



This notebook is pretty similar to the **Tutorial-ToyDataset.ipynb**. We will focus on the cells that present important changes for the understanding of the code.

One of the most important differences in this notebook, is that the ACDC dataset has four classes instead of just two. For this reason, the transformations for the mask now need to take into account the number of classes to segment. For this, we assume that the pixels in the masks have four different integer values, that we transform into numbers inside {0, 1, 2, 4}.

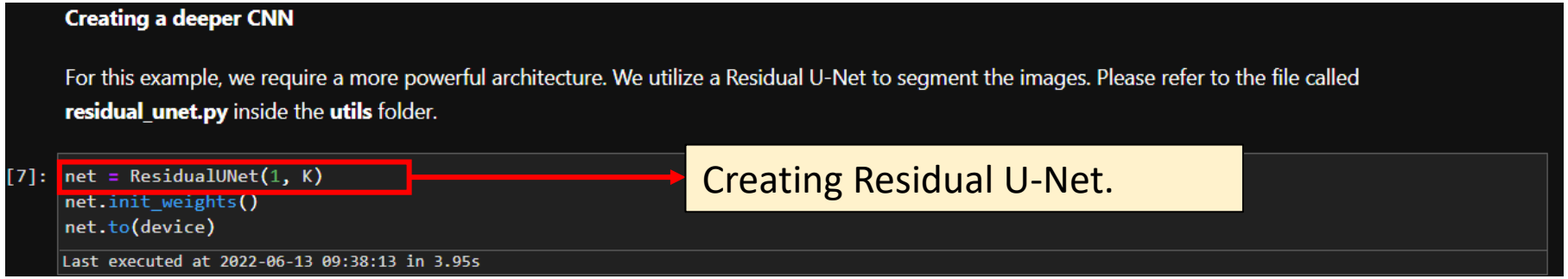


The creation of the dataset follows the same structure as before. We can also display three training samples to

In this case, we want to create a deeper CNN. Particularly, we create a Residual U-Net, which is a more powerful model for Segmentation than just a 3-layer CNN. This architecture is found in the **utils** folder, inside the file **residual_unet.py**. We simply import the file at the beginning as:

from utils.residual_unet import ResidualUNet

We can now create our network similarly as before:



After running this cell, a larger summary of the network will be printed. We can now see that this architecture has more elements as our previous model. However, if we explore such components, we can notice that they in essence the same ones as before, just arranged in a more complex way.

We now proceed to create our loss functions. We see that the only change we need is to account for more classes. This is achieved by passing more indices to the classes when we create our losses:

Creating loss functions

PyTorch have several pre-defined loss functions that we can directly use. However, there might be times when we need our own loss function, e.g. size loss. Here we show how we can create any loss function from scratch. If we construct all our objects (inputs, networks, targets, etc.) as PyTorch tensors, creating and using a loss function during backpropagation is easier. To better understand how to create customize losses, please refer to the file called "losses.py".

```
ce_loss = CrossEntropy(idk=list(range(K))) #To supervise both bac
partial_ce = CrossEntropy(idk=list(range(1, K))) #Avoid supervising backgrounc
sizeloss = NaiveSizeLoss(idk=list(range(1,K)))
```

Passing a list of indices containing the different classes.

```
Last executed at 2022-06-13 09:39:42 in 4ms
```

```
Initialized CrossEntropy with {'idk': [0, 1, 2, 3]}
Initialized CrossEntropy with {'idk': [1, 2, 3]}
Initialized NaiveSizeLoss with {'idk': [1, 2, 3]}
```

This time, we can use the same CrossEntropy class for both the full cross-entropy and partial cross-entropy losses.

Inside the training loop, we see very few changes. Particularly, we now define a more general way of calculating the prediction's size, as well as the log of the DICE coefficient and size difference. Now this generalizes to more than two classes:

```
logits = net(img)
pred_softmax = F.softmax(5 * logits, dim=1)
pred_seg = probs2one_hot(pred_softmax)

# `1:` for class 1 onward. This way handle both binary and mult-class
# 1st item, all classes except background
pred_size = einsum("bkwh->bk", pred_seg)[0, 1:].cpu()
log_sizediff[j] = (pred_size - data["true_size"][0, 1:]).mean()
log_dice[j] = dice_coef(pred_seg, full_mask)[0, 1:].mean()
```

Calculations can now be performed with more than two classes.

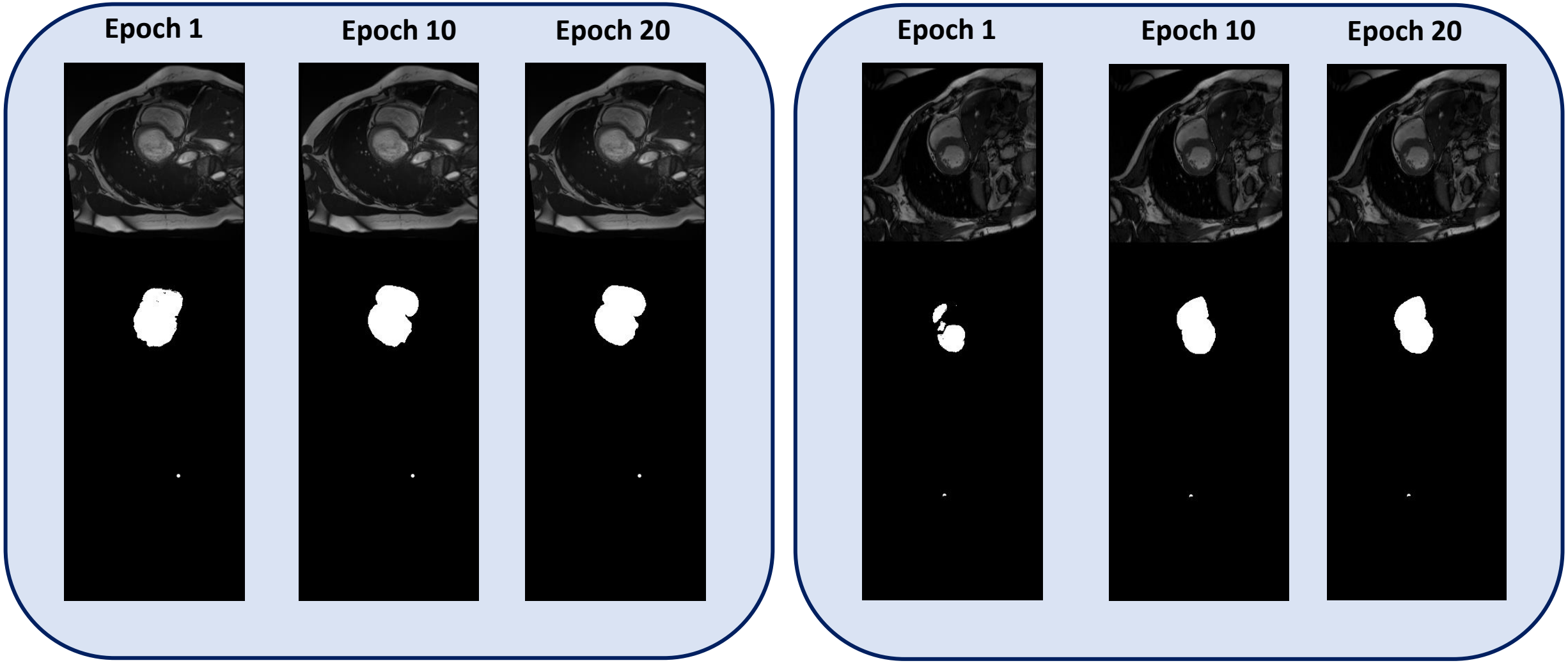
When we choose the **constrained** mode, we also see some changes in the indexing so that we can accommodate to more classes:

```
elif mode == 'constrained':
    ce_val = partial_ce(pred_softmax, weak_mask)
    log_ce[j] = ce_val.item()

    sizeLoss_val = sizeLoss(pred_softmax, bounds)
    log_sizeloss[j] = sizeLoss_val[0, 1:].mean()

    lossEpoch = ce_val + sizeLoss_val.sum() / 100
```

The training can be executed exactly as before, just changing the training mode from **full** to **unconstrained** to **constrained.** It is important to notice that this dataset takes more time per epoch, thus scaling it to the number of epochs used with the toy dataset could take more time. From now on, we show results with 20 epochs.

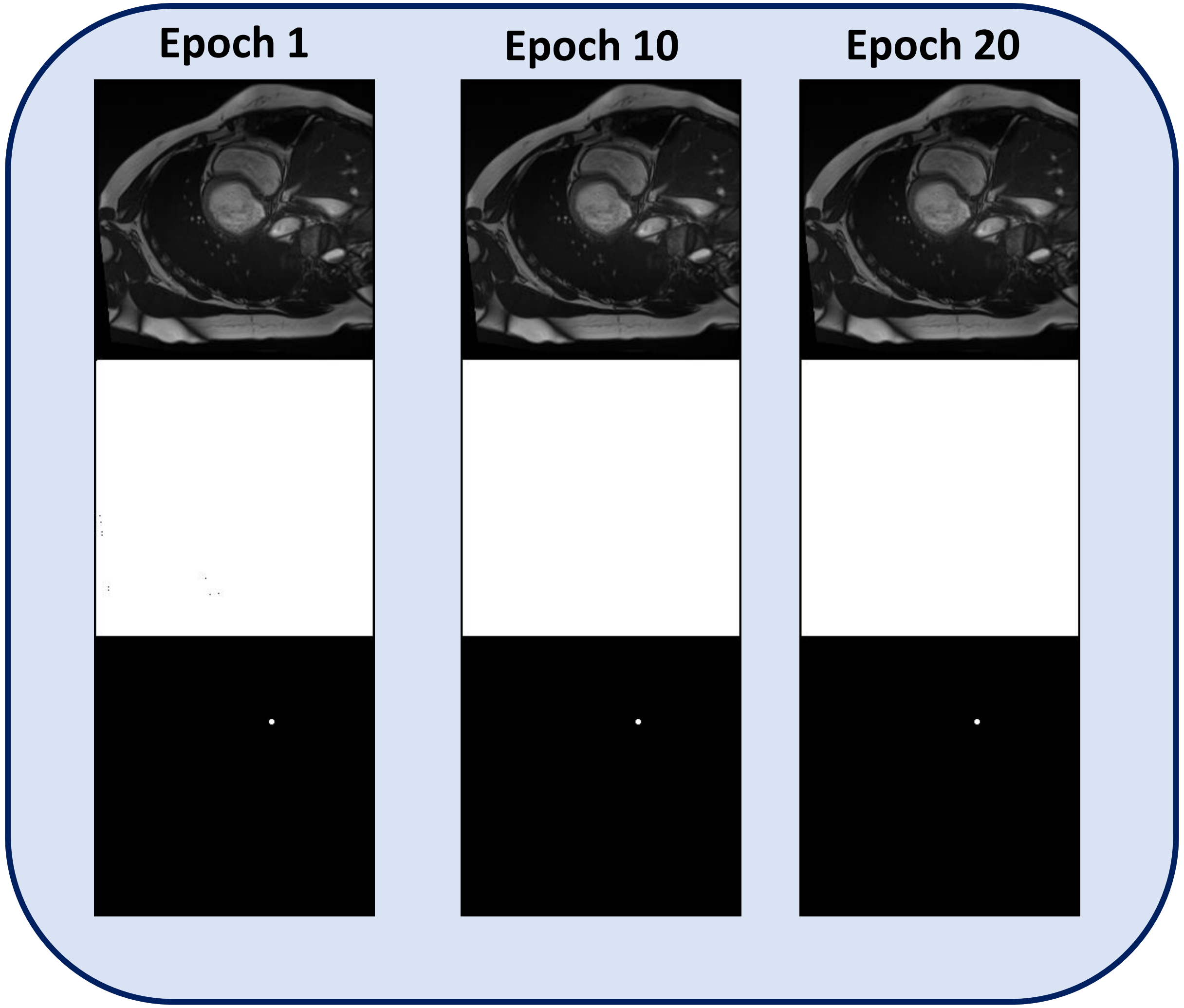
First, we execute the training with full supervision. With few epochs, and also with an initial implementation as the one we utilize, we cannot expect state-of-the-art results. However, we are able to observe the effects of the constrained CNN, and also the method can be built on in order to improve the results. We encourage the assistants of this tutorial to run the training of the model for longer, so that better results can be appreciated.

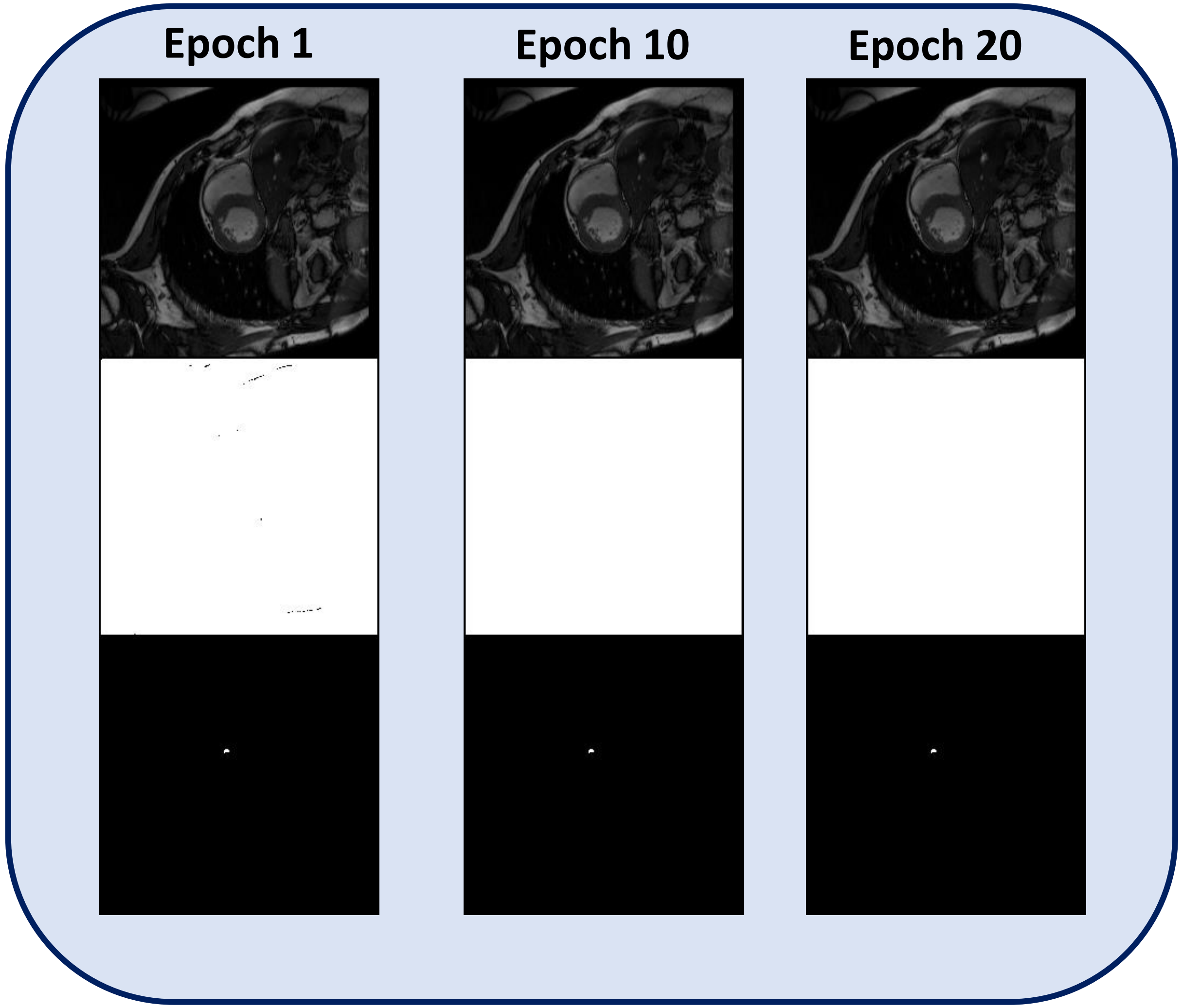
```
>> Training   (  0): 100%| | 1298/1298 [ 9.99it/s, DSC=0.738, SizeDiff=-0012.6, LossCE=2.70e-02]
>> Training   ( 19): 100%| | 1298/1298 [ 9.90it/s, DSC=0.852, SizeDiff=00002.0, LossCE=1.12e-02]
```



Now we execute the experiment with only the partial cross-entropy loss, without using any constrain. As expected, the network is not able to learn as very little supervision without auxiliar priors is not useful for segmentation.

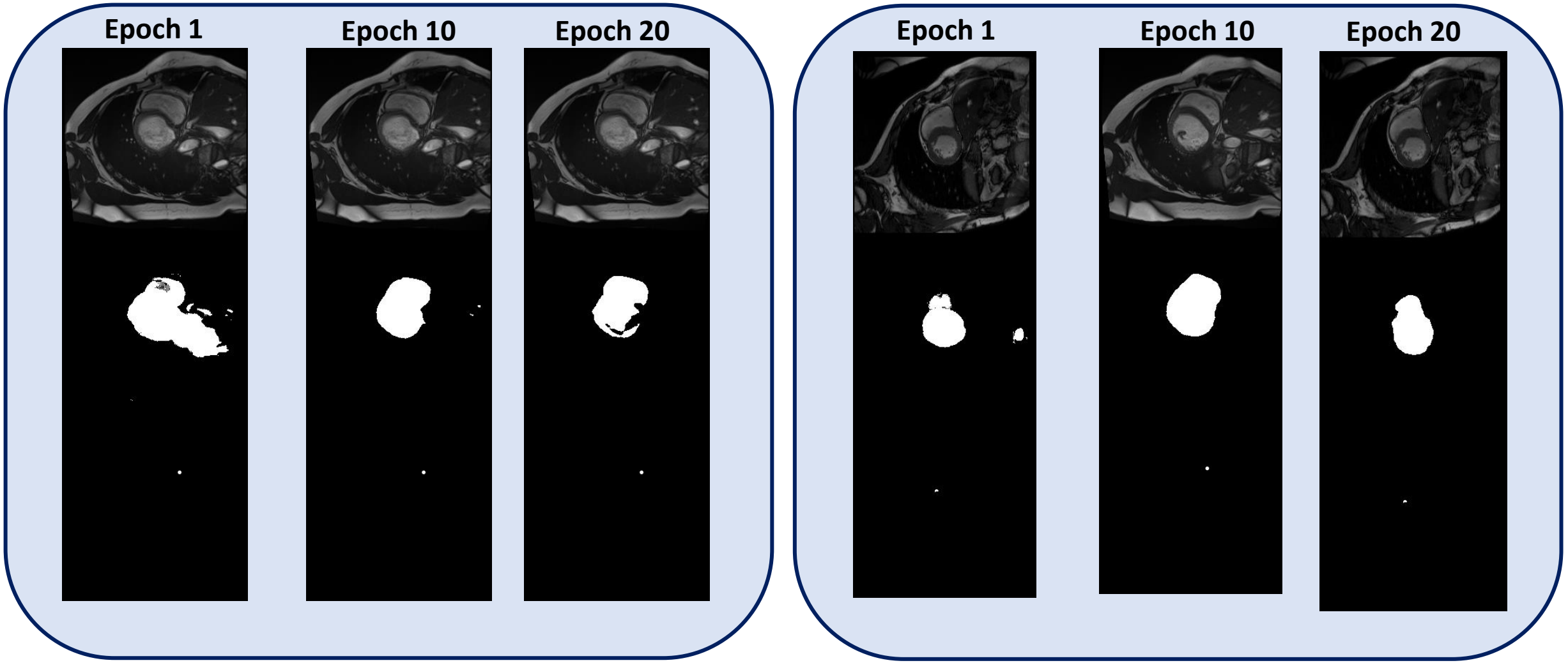
```
>> Training   (   0): 100%|████████████████████| 1298/1298 [ 9.79it/s, DSC=0.052, SizeDiff=19679.4, LossCE=1.26e+00]
>> Training   (  19): 100%|████████████████████| 1298/1298 [ 9.80it/s, DSC=0.120, SizeDiff=21033.0, LossCE=1.74e-01]
```





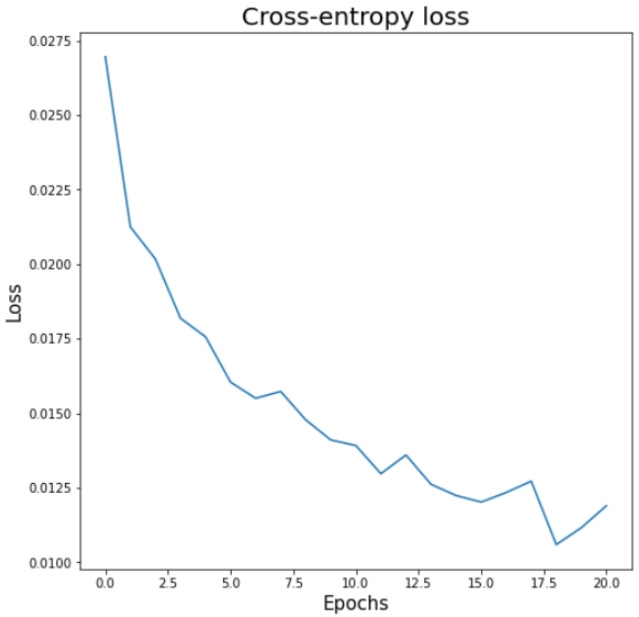
Now we execute the experiment with only the partial cross-entropy loss, without using any constrain. As expected, the network is not able to learn as very little supervision without auxiliar priors is not useful for segmentation.

```
>> Training   (   0): 100%| | 1298/1298 [ 9.72it/s, DSC=0.270, SizeDiff=02473.1, LossCE=2.18e+00, LossSize=5.49e+02]
>> Training   (  20): 100%| | 1298/1298 [ 9.63it/s, DSC=0.722, SizeDiff=00180.2, LossCE=1.41e-01, LossSize=8.28e-01]
```

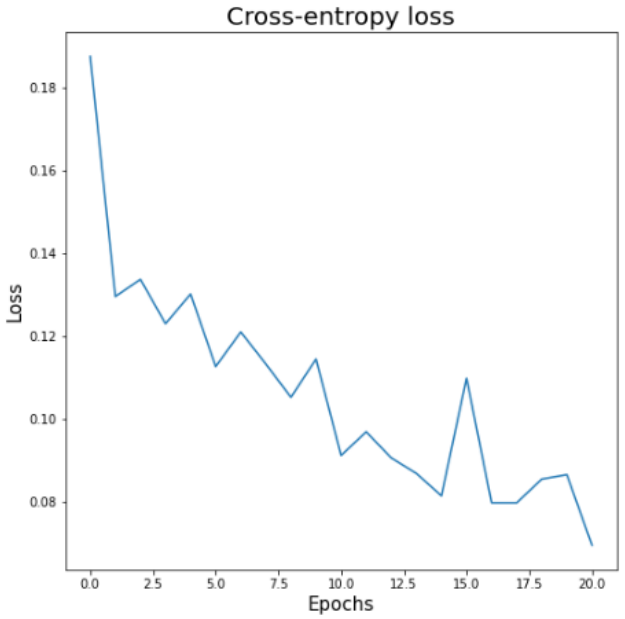


Once again, we can plot the cross-entropy loss in the three scenarios to observe the effects of the training of the model using full labels, weak labels without constraints, and constraining the loss.

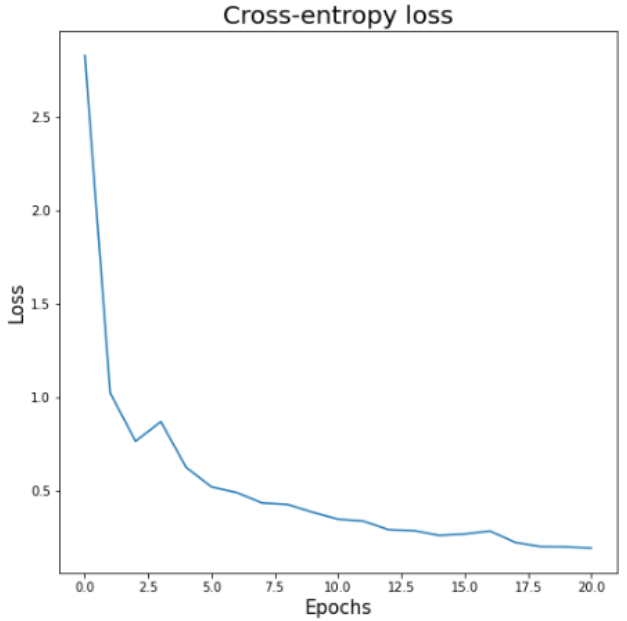
```
[15]: '''Displaying some examples'''
      fig = plt.figure(0)
      fig.set_figheight(8)
      fig.set_figwidth(8)

      plt.plot(np.arange(epochs), losses)

      plt.title('Cross-entropy loss', fontsize=20)
      plt.xlabel('Epochs', fontsize=15)
      plt.ylabel('Loss', fontsize=15)
      plt.show()
      Last executed at 2022-06-14 18:32:17 in 118ms
```



Fully supervision



Weak supervision



Weak supervision + constraint