Published in Towards Data Science

Sagi eppel Follow

Jun 2, 2022 · 8 min read · Listen

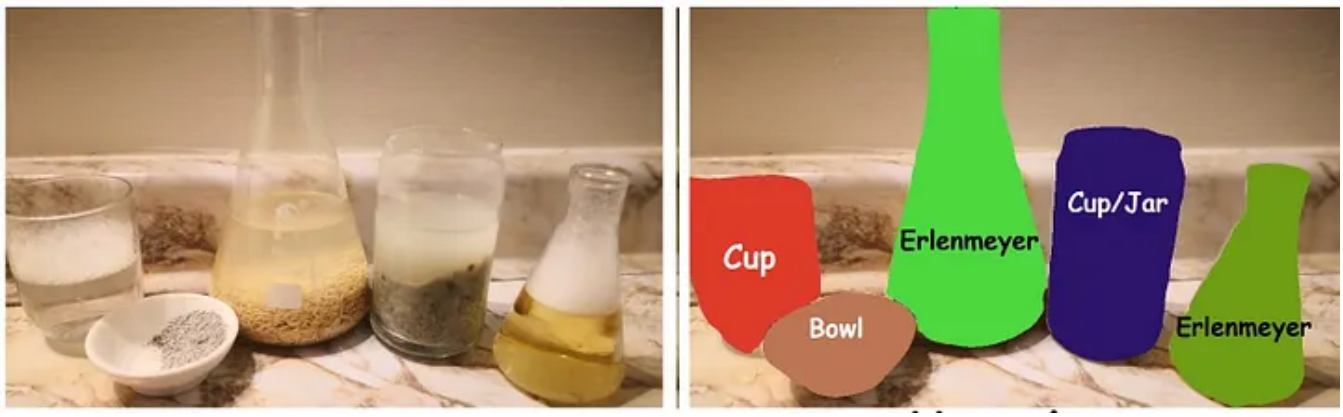


# Train Mask R-CNN Net for Object Detection in 60 Lines of Code

## A step-by-step tutorial using a minimal amount of code

Object detection and instance segmentation is the task of identifying and segmenting objects in images. This involves finding for each object the bounding box, the mask that covers the exact object, and the object class. Mask R-CNN is one of the most common methods to achieve this.

This tutorial aims to explain how to train such a net with a minimal amount of code (60 lines not including spaces).




Example for object detection/instance segmentation. The image appears on the left and the objects and their classes appear to the right.

Code is available at: https://github.com/sagieppel/Train_Mask-RCNN-for-object-detection-in_In_60_Lines-of-Code

The project will use Pytorch 1.1 and OpenCV packages.

Note that the PyTorch MaskRCNN implementation might have some issues with the newer PyTorch versions, so PyTorch 1.1 is a safe option.

Pytorch installation instructions are available at:

https://pytorch.org/get-started/locally/

OpenCV can be installed using:

```
pip install opencv-python
```

Next, we need a dataset for training. We will use the LabPics V2 dataset that can be downloaded from here:

https://zenodo.org/record/4736111/files/LabPicsChemistry.zip?download=1

The dataset is free to use under MIT license.

We will train the net to detect all the vessels in the images.

Now we can start writing the code.

First, let's import packages and define the main training parameters:

```
import random
from torchvision.models.detection.faster_rcnn import
FastRCNNPredictor
import numpy as np
import torch.utils.data
import cv2
import torchvision.models.segmentation
import torch
import os

batchSize=2
imageSize=[600,600]

device = torch.device('cuda') if torch.cuda.is_available() else
torch.device('cpu')
```

*imageSize=[Width,height]* are the dimensions of the image used for training. All images during the training processes will be resized to this size.

*batchSize: i*s the number of images that will be used for each iteration of the training.

*batchSize*Width*Height* will be proportional to the memory requirement of the training. Depending on your hardware, it might be necessary to use a smaller batchSize or image size to avoid out-of-memory problems.

Note that since we train with only a single image size, the net once trained is likely to be limited to work with only images of this size. In most cases what you want to do is change the size of each training batch.

*device:* automatically set the device where the net will run (GPU or CPU), in practice training without a strong GPU is extremely slow.

Next, we create a list of all images in the dataset:

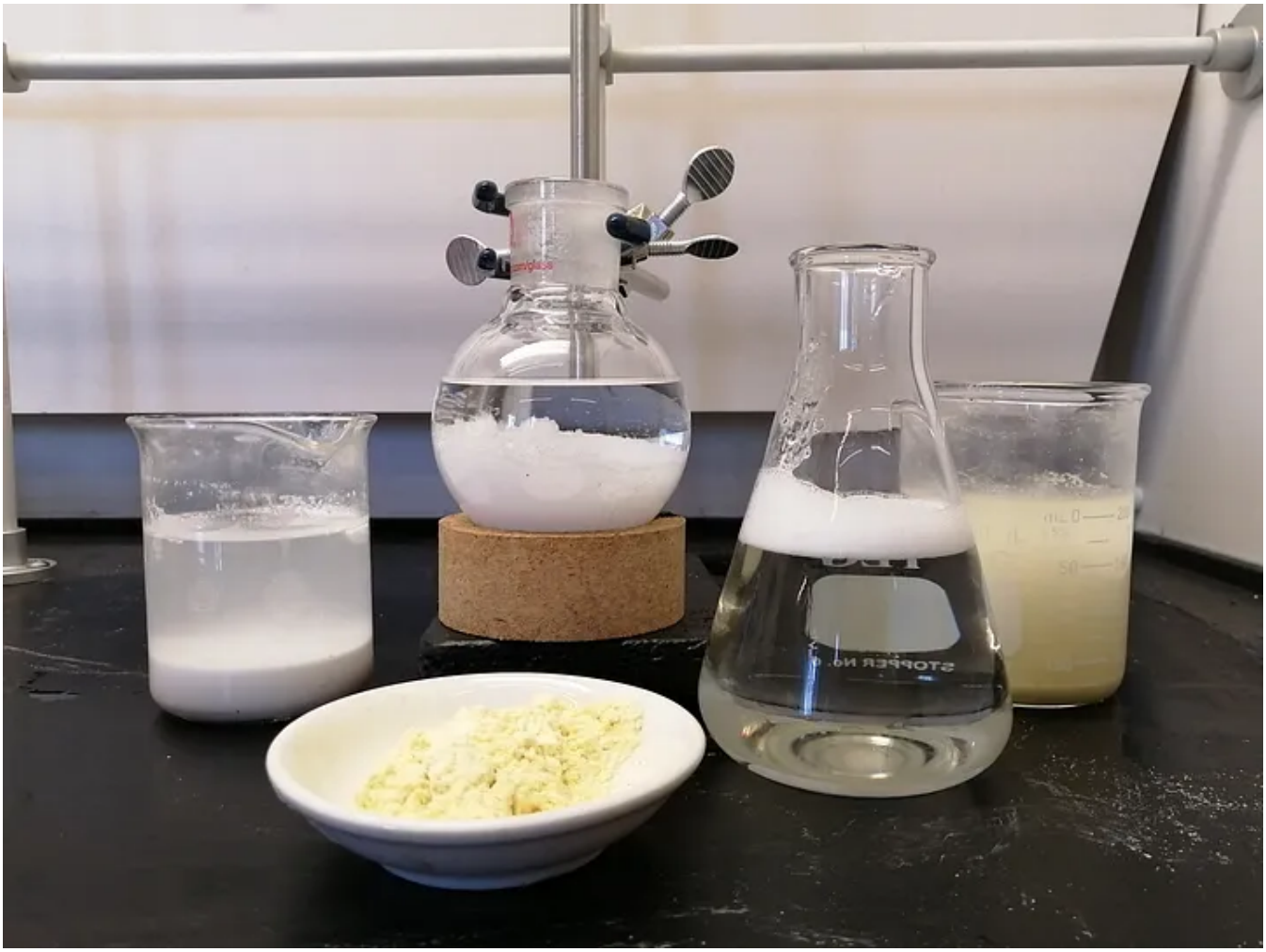
```
trainDir="LabPicsChemistry/Train"

imgs=[]
for pth in os.listdir(trainDir):
    imgs.append(trainDir+"/"+pth +"//")
```

*TrainDir:* is the LabPics V2 dataset train folder.

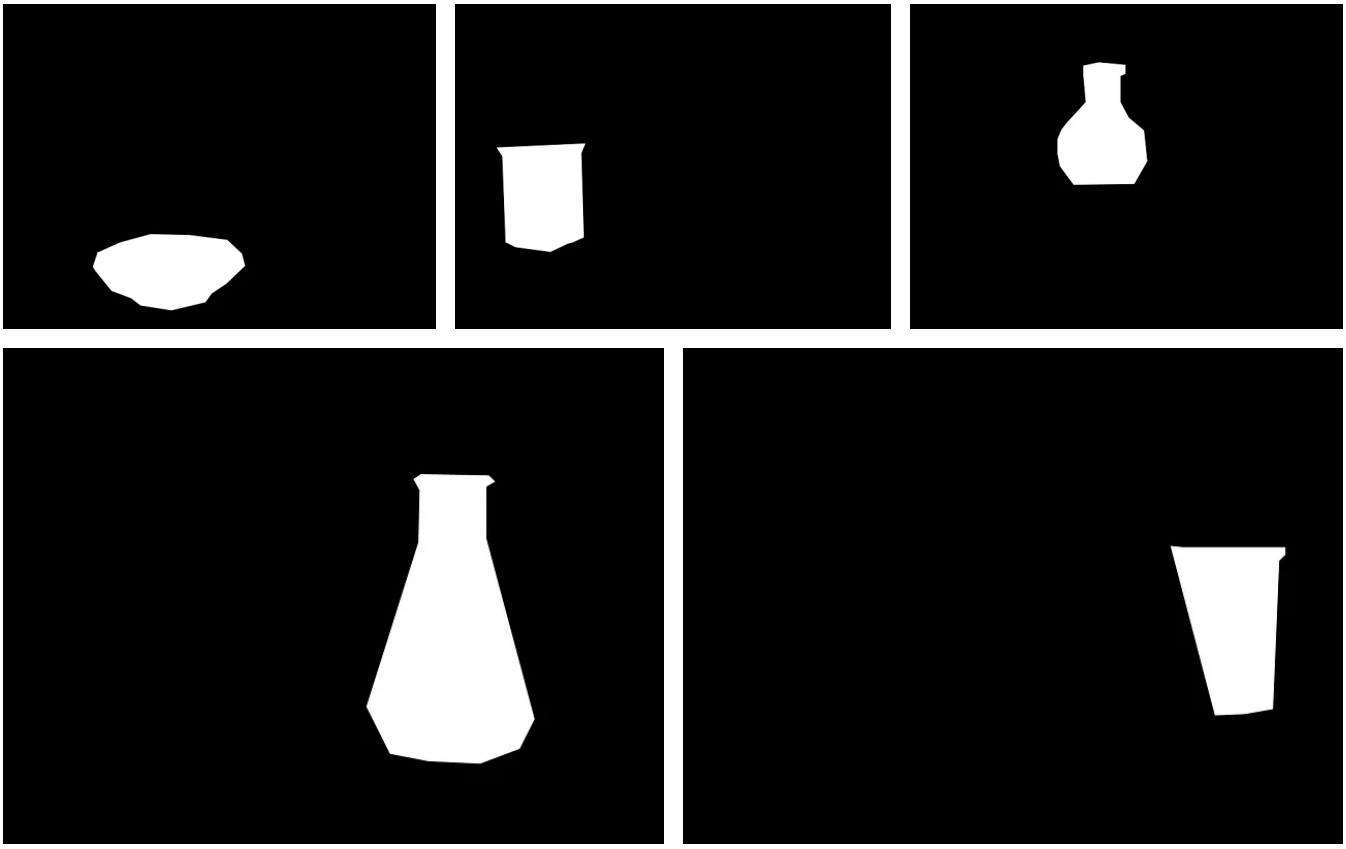
imgs: is the list of all images in the trainset

Next, we create a data loader function that will allow us to load a batch of random images and their data for training. The data will contain the image:



Image

and masks of all the objects in the image. Each mask will be saved as a black-white (0/1) image:



Masks of different vessels in the image.

The full data loader function code is:

```
def loadData():
  batch_Imgs=[]
  batch_Data=[]
  for i in range(batchSize):

        idx=random.randint(0,len(imgs)-1)
        img = cv2.imread(os.path.join(imgs[idx], "Image.jpg"))
        img = cv2.resize(img, imageSize, cv2.INTER_LINEAR)

        maskDir=os.path.join(imgs[idx], "Vessels")
        masks=[]
        for mskName in os.listdir(maskDir):
            vesMask = cv2.imread(maskDir+'/'+mskName, 0)
            vesMask = (vesMask > 0).astype(np.uint8)
            vesMask=cv2.resize(vesMask,imageSize,cv2.INTER_NEAREST)
            masks.append(vesMask)

        num_objs = len(masks)
        if num_objs==0: return loadData()

        boxes = torch.zeros([num_objs,4], dtype=torch.float32)
        for i in range(num_objs):
            x,y,w,h = cv2.boundingRect(masks[i])
```

```
            boxes[i] = torch.tensor([x, y, x+w, y+h])
        masks = torch.as_tensor(masks, dtype=torch.uint8)
        img = torch.as_tensor(img, dtype=torch.float32)

        data = {}
        data["boxes"] =  boxes
        data["labels"] =  torch.ones((num_objs,), dtype=torch.int64)
        data["masks"] = masks

        batch_Imgs.append(img)
        batch_Data.append(data)

    batch_Imgs=torch.stack([torch.as_tensor(d) for d in batch_Imgs],0)
    batch_Imgs = batch_Imgs.swapaxes(1, 3).swapaxes(2, 3)
    return batch_Imags, batch_Data
```

What this function does is load a set of images, for each image it loads a set of masks for the vessels in the image. It then generates bounding boxes and classes for each object.

Let's look at this function line by line:

```
idx=random.randint(0,len(imgs)-1)
img = cv2.imread(os.path.join(imgs[idx], "Image.jpg"))
img = cv2.resize(img, imageSize, cv2.INTER_LINEAR)
```





Next, we load the object's masks. These masks are images the same size as the RGB image where the region of the object instances is marked 1 and the rest are marked 0.

```
maskDir=os.path.join(imgs[idx], "Vessels")
masks=[] # list of all object mask in the image
for mskName in os.listdir(maskDir):
            vesMask = cv2.imread(maskDir+'/'+mskName, 0) # Read mask
```

```
            vesMask = (vesMask > 0).astype(np.uint8) #convert to 0-1
            vesMask=cv2.resize(vesMask,imageSize,cv2.INTER_NEAREST)
            masks.append(vesMask) # add to the mask list
```

*maskDir:* is the subfolder where the vessel instances map is stored.

We first read the masks.

```
vesMask = cv2.imread(maskDir+'/'+mskName, 0)
```

The mask image is stored in 0–255 format and is converted to 0–1 format:

```
vesMask = (vesMask > 0).astype(np.uint8)
```

Finally, the mask is resized to the standard image size and added to the mask list.

```
vesMask=cv2.resize(vesMask,imageSize,cv2.INTER_NEAREST)
masks.append(vesMask)
```



Next, we use the masks to generate a bounding box for each object.

```
boxes = torch.zeros([num_objs,4], dtype=torch.float32)
for i in range(num_objs):
            x,y,w,h = cv2.boundingRect(masks[i])
            boxes[i] = torch.tensor([x, y, x+w, y+h])
```

Luckily OpenCV has the function that generates a bounding box from mask:

```
x,y,w,h = cv2.boundingRect(masks[i])
```

*x,y:* are the top coordinate of the bounding box.

*w,h:* are the width and height of the bounding box.

The mask RCNN bounding box format demands the top left and bottom right coordinate of the box which is given by: *[x, y, x+w, y+h].*

```
boxes[i] = torch.tensor([x, y, x+w, y+h])
```

Finally, we stack all the information about the image into one dictionary.

```
data = {}
data["boxes"] =  boxes
data["labels"] =  torch.ones((num_objs,), dtype=torch.int64)
data["masks"] = masks
```

Note that for labels we just pick ones for everything. That means that we just take the class of all the objects to be the same (1).

And we are done. We just need to load the image data into the training batch and convert it to PyTorch format.

```
        batch_Imgs.append(img)
        batch_Data.append(data)
batch_Imgs= torch.stack([torch.as_tensor(d) for d in batch_Imgs], 0)
batch_Imgs = batch_Imgs.swapaxes(1, 3).swapaxes(2, 3)
feturn batch_Imags, batch_Data
```

*batch_Imgs, batch_Imgs:* are the batch (list) of images used for training and their data.

Now that we have the data loader we can start building the net. First, we load a mask RCNN model that was already pretrained on the COCO dataset:

```
model=torchvision.models.detection.maskrcnn_resnet50_fpn(pretrained=
True)
```

A pretrained model that uses existing knowledge can learn new tasks and datasets much faster than a model that was not trained before.

The COCO dataset contains over 100 classes. In our case, we only need two classes. We will therefore change the final layers of the net to predict two classes:

```
in_features = model.roi_heads.box_predictor.cls_score.in_features
model.roi_heads.box_predictor=FastRCNNPredictor(in_features,num_clas
ses=2)
```

Finally, we load the model to the training device GPU or CPU:

```
model.to(device)
```

We can now define the optimizer which will determine the way the net weights will be changed during training:

```
optimizer = torch.optim.AdamW(params=model.parameters(), lr=1e-5)
```

We use AdamW which is the latest optimizer and has a standard learning rate of *lr=1e-5*

We set the model to train mode:

```
model.train()
```

And we are ready to write the main training loop:

```
for i in range(10001):
   images, targets = loadData()
   images = list(image.to(device) for image in images)
   targets=[{k: v.to(device) for k,v in t.items()} for t in targets]

   optimizer.zero_grad()
   loss_dict = model(images, targets)
   losses = sum(loss for loss in loss_dict.values())

   losses.backward()
   optimizer.step()

   print(i,'loss:', losses.item())
   if i%200==0:
           torch.save(model.state_dict(), str(i)+".torch")
           print("Save model to:",str(i)+".torch")
```

We are training for 10001 iterations.

The first part is loading the data using the data loader function we just define:

```
images, targets = loadData()
```

Next, we load the data into the training device (CPU/GPU)

```
images = list(image.to(device) for image in images)
targets=[{k: v.to(device) for k,v in t.items()} for t in targets]
```

We take the images and data and run it through our neural net to get the loss:

```
loss_dict = model(images, targets)
```

The loss is composed of several parts: class loss, bounding box loss, and mask loss. We sum all of these parts together to get the total loss as a single number:

```
losses = sum(loss for loss in loss_dict.values())
```

Once we have the loss we can update the neural net weights using backpropagation.

```
losses.backward()
optimizer.step()
```

We want to save our trained model once every 500 steps so we can use it later.

```
if i%500==0:
    torch.save(model.state_dict(), str(i)+".torch")
```

And that's it. After 4-10k training iteration, we should start getting good results.

The full code can be found here: https://github.com/sagieppel/Train_Mask-RCNN-for-object-detection-in_In_60_Lines-of-Code/blob/main/train.py

Once we finished the training we want to test our model.

For this, we will use the evaluation script available here: https://github.com/sagieppel/Train_Mask-RCNN-for-object-detection-in_In_60_Lines-of-Code/blob/main/test.py

The script is similar to the training script. The first part is simply loading the net as before.

```
device = torch.device('cuda') if torch.cuda.is_available() else
torch.device('cpu')
model=torchvision.models.detection.maskrcnn_resnet50_fpn(pretrained=
True)
in_features = model.roi_heads.box_predictor.cls_score.in_features
model.roi_heads.box_predictor=FastRCNNPredictor(in_features,num_clas
ses=2)

model.load_state_dict(torch.load("10000.torch"))
model.to(device)# move model to the right devic
model.eval()
```

The only difference is we also add one line to load the saved model, and set the model to evaluation state instead of training state:

```
model.load_state_dict(torch.load("10000.torch"))
model.eval()
```

Next, we load a single image, resize it to standard size, and convert it to PyTorch format:

```
images = cv2.imread(imgPath)
images = cv2.resize(images, imageSize, cv2.INTER_LINEAR)
images = torch.as_tensor(images, dtype=torch.float32).unsqueeze(0)
images=images.swapaxes(1, 3).swapaxes(2, 3)
images = list(image.to(device) for image in images)
```

We run the image through the net:

```
with torch.no_grad():
    pred = model(images)
```

This runs the image through the net and gets a prediction for the object in the image. Note we are not training the net, so we do not need to collect gradient (no_grad) this makes the net run much faster.

The prediction is composed of several parts: “masks” which corresponds to the mask (regions) of every object in the image. “Scores” correspond to how likely the predicted mask is correct. In addition, we have the predicted bounding box and classes but we will not use those.

We will go over all the predictions and display only those objects with “scores” larger than 0.8:

```
im= images[0].swapaxes(0, 2).swapaxes(0,
1).detach().cpu().numpy().astype(np.uint8)

im2 = im.copy()

for i in range(len(pred[0]['masks'])):
    msk=pred[0]['masks'][i,0].detach().cpu().numpy()
    scr=pred[0]['scores'][i].detach().cpu().numpy()
    if scr>0.8 :
        im2[:,:,0][msk>0.5] = random.randint(0,255)
        im2[:, :, 1][msk > 0.5] = random.randint(0,255)
        im2[:, :, 2][msk > 0.5] = random.randint(0, 255)
cv2.imshow(str(scr), np.hstack([im,im2]))
cv2.waitKey()
```

Note that the predicted object 'masks' are saved as a matrix in the same size as the image with each pixel having a value that corresponds to how likely it is part of the object.

We can assume that only pixels which values larger than 0.5 are likely to be part of the objects. We display this by marking these pixels with a different random color for each object:

```
im2[:,:,0][msk>0.5] = random.randint(0,255)
im2[:, :, 1][msk > 0.5] = random.randint(0,255)
im2[:, :, 2][msk > 0.5] = random.randint(0, 255)
```

The results can be seen here:

Original Image to the left and predicted object regions to the right.

The full testing code can be found here:

**Train_Mask-RCNN-for-object-detection-in_In_60_Lines-of-Code/test.py at main ·...**

This file contains bidirectional Unicode text that may be interpreted or compiled differently than what appears below...

github.com

Tutorial for mask RCNN with PyTorch:

https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html

Original mask RCNN paper can be downloaded from here:

Machine Learning Computer Vision Object Detection Pytorch Deep Learning