CSGI-GA-3303076 Vision meets ML

Homework 3 Part 1

Enter your name and NetID below.

Name: Shuvadeep Saha

NetID: ss15592

The main goals of this assignment include:

1. Giving an introduction to Mask-RCNN
2. Training the predictors for a given dataset
3. Finetuning the entire network for the same dataset
4. Building an entire segmentation pipeline

There are **TWO PARTS** in this assignment.

Part1: Accompanying each part, there are a few questions (**12 questions in total**) -- 10 mandatory + 2 extra credit. The first 10 questions are worth 100 points and the extra credit questions are worth 20 points.

Part2: Please use HW3Part2.ipynb for part 2. Merge the pdf exports of both notebook and submit **1 PDF**.

Please give your answers in the space provided. This homework has a mix of conceptual and coding questions. You can quickly navigate to coding questions by searching (Ctrl/Cmd-F) for `TODO:` .

Part 1: Introduction to Mask-RCNN

[Mask-RCNN (https://arxiv.org/pdf/1703.06870.pdf)](https://arxiv.org/pdf/1703.06870.pdf) is a network used for instance segmentation. Instance segmentation can be thought of as a hybrid of semantic segmentation and object detection. In other words, we don't want to just find the bounding boxes for each object in our image, we're also interested in finding the segmentation mask of *each object instance*.

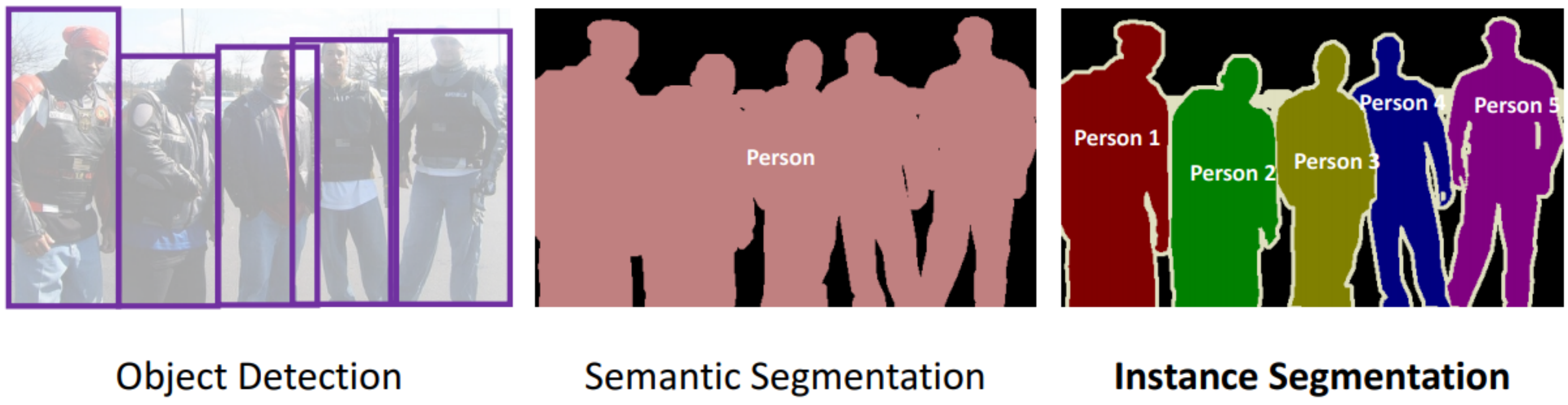


Image Credits: [https://towardsdatascience.com/single-stage-instance-segmentation-a-review-1eeb66e0cc49 (https://towardsdatascience.com/single-stage-instance-segmentation-a-review-1eeb66e0cc49)](https://towardsdatascience.com/single-stage-instance-segmentation-a-review-1eeb66e0cc49)

Mask-RCNN is built on top of Faster-RCNN, which is a network used for object detection. Faster-RCNN has 2 outputs for each candidate object (Region of Interest or RoI) - a class label and a bounding box offset. Mask-RCNN adds a third branch to Faster-RCNN for predicting segmentation masks on each RoI.

We'll first briefly go over Faster-RCNN. Faster-RCNN has 2 stages:

1. **Region Proposal Network (RPN):** Given the image, it proposes candidate object bounding boxes. Previous object detection models such as RCNN and Fast-RCNN handled this separately from the CNN model. Faster-RCNN takes a different approach -- it integrates these two components into the same network to achieve speedup.
2. **Fast-RCNN:** This stages takes each candidate RoI and extracts features from the image feature vector using RoIPool. Using these RoI features, it performs classification and bounding box regression.

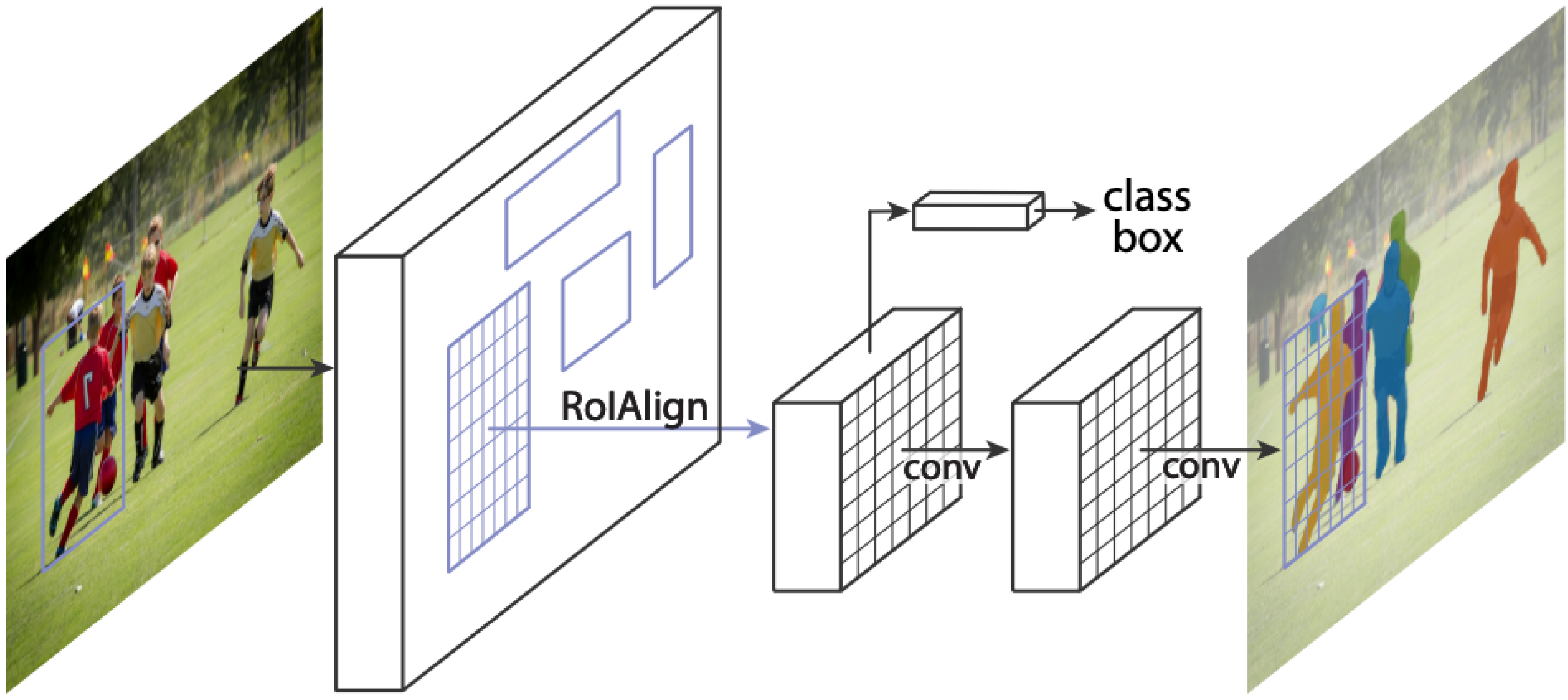


Figure 1. The **Mask R-CNN** framework for instance segmentation.

Mask-RCNN has the same 2-stage procedure, but in the 2nd stage, instead of just predicting the classification label and the bounding box offset, it predicts **in parallel** a binary mask for each RoI.

Mask-RCNN relies on a pretrained network (called the "backbone" in the paper) to extract features from the image. These features are fed into the Region Proposal Network (RPN) to generate candidate RoIs. For each RoI candidate, a fixed size RoI feature vector is generated using an RoIAlign layer. This RoI feature map is then provided to the classifier, bounding box predictor and the segmentation mask to generate the final output.

Question 1

Training uses a multi-task loss function. What are the three components in this loss function? Is the loss computed per image or per RoI?

Answer:

The multi-class loss function is defined, during training on each RoI. The three components of the loss function are: Classification loss, Bounding-box regression loss and average binary cross entropy loss.

Question 2

[This blog post (https://lilianweng.github.io/lil-log/2017/12/31/object-recognition-for-dummies-part-3.html)](https://lilianweng.github.io/lil-log/2017/12/31/object-recognition-for-dummies-part-3.html) by Lilian Weng gives a nice overview of the object detection (RCNN type) networks. In the blog post, it is mentioned that Mask-RCNN uses RoIAlign instead of RoIPool. Explain briefly in 3-4 lines why this is being done.

Answer

RoIPool first quantizes a floating-number RoI to the discrete granularity of the feature map, this quantized RoI is then subdivided into spatial bins which are themselves quantized, and finally feature values covered by each bin are aggregated (usually by max pooling). Though this does not affect the classification, which is robust to small translations, it has a large negative effect on predicting pixel-accurate masks.

To avoid this, Mask-RCNN uses RoIAlign which use bi-linear interpolation to compute the exact values of the input features at four regularly sampled locations in each RoI bin, and aggregate the result using max or average pooling. This gives pixel-accurate masks for image segmentation.

Question 3

What are the different backbones explored in the Mask-RCNN paper? They are denoted in the paper using network-depth-features nomenclature. What is the advantage of using a ResNet-FPN backbone over a ResNet-C4 backbone for feature extraction?

Answer

The authors evaluated ResNet, ResNeXt networks of depth 50 or 101 layers and FPN (Feature Pyramid Network) as the backbones in the paper.

Using a ResNet-FPN backbone for feature extraction with Mask RCNN gives excellent gains in both accuracy and speed.

Part 2: Training the Predictors for a New Dataset

In this section, we'll start with a pretrained Mask-RCNN model that uses Resnet-50-FPN as the backbone. This model was trained on MS-COCO dataset which is widely used for multiple vision tasks such as object detection, instance segmentation, etc.

MS-COCO has 91 classes (90 for objects + 1 for background). Some sample objects in the dataset include `person` , `car` , `bicycle` , `knife` , `train` , etc.

Along with this homework file, we have also provided another sample dataset (we'll refer to it as the [Nature dataset (https://towardsdatascience.com/custom-instance-segmentation-training-with-7-lines-of-code-ff340851e99b)](https://towardsdatascience.com/custom-instance-segmentation-training-with-7-lines-of-code-ff340851e99b)). It isn't a standard dataset, but it's small enough (600 train + 200 test images) and allows us to easily demo finetuning a pretrained Mask-RCNN model. This dataset contains only 2 classes - `squirrel` and `butterfly` .

Our goal in part2 and part3 of this assignment is to take the pretrained Mask-RCNN model and finetune/train it for this dataset. However, here in part2, instead of finetuning the entire network, we'll train only the final layers.

In Homework 2, we've shown how one could feed data into the network using `Dataset` s and `DataLoader` s. We'll use the same strategy here for finetuning the model.

We've based this homework on this [PyTorch tutorial on Object Detection Finetuning (https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html)](https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html).

In [1]:

```
import numpy as np
import torch
import torchvision

import json # for reading from json file
import glob # for listing files inside a folder
from PIL import Image, ImageDraw # for reading images and drawing masks on them.


# Create a custom dataset class for Nature
# dataset subclassing PyTorch's Dataset class
class NatureDataset(torch.utils.data.Dataset):
    def __init__(self, root, transforms):
        self.transforms = transforms

        # Load all image files, sorting them to
        # ensure that they are aligned with json files
        imgs = glob.glob(root + '/*.jpg')
        imgs += glob.glob(root + '/*.png') # some images are in png format
        self.imgs = sorted(imgs)

        # Mask data is stored in a json file
        masks = glob.glob(root + '/*.json')
        self.masks = sorted(masks)

        # Each image can have multiple object instances, and each
        # instance is associated with either of these 2 labels.

        # Need to convert str-labels to ids. So we'll use
        # this label-to-index mapping.
        # Note: we can't start from 0 because 0 is restricted
        # to the "background" class
        self.label_to_id = {'squirrel': 1, 'butterfly': 2}


    def __getitem__(self, idx):
        # Have already aligned images and JSON files; can now
        # simply use the index to access both images and masks
        img_path = self.imgs[idx]
        mask_path = self.masks[idx]

        # Read image using PIL.Image and convert it to an RGB image
        img = Image.open(img_path).convert("RGB")

        # TODO: Read image height, width and mask data from
        # the JSON file
        with open(mask_path, 'r') as fp:
            # TODO: Using json library read the dictionary
            # from the fp
            json_dict = json.load(fp)

            # TODO:
            height = json_dict['imageHeight']

            # TODO:
            width = json_dict['imageWidth']

            # TODO:
            poly_shapes_data = json_dict["shapes"]

        # TODO: Each image can have multiple mask instances.
        # Using the polygon points, generate the 2d-mask
        # using PIL's ImageDraw.polygon
        masks = []
        labels = []
        for shape_data in poly_shapes_data:
            polygon_points = [tuple(point) for point in shape_data['points']]

            # TODO: Using Image.new() create an image of size (width, height)
            # and fill it with 0s.
            mask_img = Image.new(mode="L", size=(width, height))

            # TODO: Draw the mask on the base image we just created
            ImageDraw.Draw(mask_img).polygon(polygon_points, outline=1, fill=1)

            mask = np.array(mask_img)
            masks.append(mask)

            label = shape_data['label']
            labels.append(label)


        # Each mask instance also has an associated label which is str-type
        # Convert the str into an int using the mapping we created in __init__
        labels = [self.label_to_id[label] for label in labels]

        # TODO: Generate the bounding boxes for each instance
        # from the 2d masks
        num_objs = len(masks)

```

```
        boxes = []
        for i in range(num_objs):
            # TODO: Use np.where() to find where mask[i] == True.
            # pos will be a 2d-list of indices
            pos = np.where(masks[i]==True)

            # In pos, find the min x- and y- indices;
            # max x- and y- indices. This will give us our box bounds.

            # TODO:
            xmin = min(pos[1])

            # TODO:
            xmax = max(pos[1])

            # TODO:
            ymin = min(pos[0])

            # TODO:
            ymax = max(pos[0])

            boxes.append([xmin, ymin, xmax, ymax])


        # Convert everything into a torch.Tensor
        boxes = torch.as_tensor(boxes, dtype=torch.float32)
        labels = torch.as_tensor(labels, dtype=torch.int64)
        masks = torch.as_tensor(masks, dtype=torch.uint8)

        image_id = torch.tensor([idx])
        area = (boxes[:, 3] - boxes[:, 1]) * (boxes[:, 2] - boxes[:, 0])

        # Assume all instances are not crowd
        iscrowd = torch.zeros((num_objs,), dtype=torch.int64)

        target = {}
        target["boxes"] = boxes
        target["labels"] = labels
        target["masks"] = masks
        target["image_id"] = image_id
        target["area"] = area
        target["iscrowd"] = iscrowd

        # Apply transforms
        if self.transforms is not None:
            img, target = self.transforms(img, target)

        return img, target


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

```
/opt/conda/envs/py37/lib/python3.7/site-packages/tqdm/auto.py:22: TqdmWarning: IProgress not found. Please upd
ate jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html (https://ipywidg
ets.readthedocs.io/en/stable/user_install.html)
  from .autonotebook import tqdm as notebook_tqdm
```

Question 4

Complete the TODO sections in the above block. Specifically:

1. Read image height, width and polygon shapes data from the JSON file.
2. Generate 2D masks from the polygon points. You can follow this idea: https://stackoverflow.com/a/3732128 (https://stackoverflow.com/a/3732128)
3. Generate the bounding boxes from the mask data. Assume that the bounding box is a rectangle with the smallest area enclosing the mask.

You may find this json schema useful:

```
{
   "shapes": [ # list of object instances; masks are represented as polygons
       ## data for instance1
       {
           "label": "" # label for instance1
           "points": [] # 2d list of polygon points [(x1, y1), (x2, y2), ..]
       },
       ## data for instance2
       {
           "label": []
           "points": []
       },

       ..
       ..
   ],
   "imagePath": ""
   "imageData" : ""
   "imageHeight": <integer>
   "imageWidth": <integer>
}
```

In [2]:

```python
# Alternatively, you could also uncomment the line below to see a sample json file
#!cat '../shared/datasets/nature-dataset/train/s (3).json'
```

Having implemented our `NatureDataset` class, let's create the `Dataset` and the `DataLoader` objects. Note that we're not using `torchvision.transforms` , instead we're using transforms provided in a separate script in this directory called `transforms.py` .

In [3]:

```python
import transforms as T

def get_transform(train):
    transforms = []
    transforms.append(T.ToTensor())
    if train:
        transforms.append(T.RandomHorizontalFlip(0.5))
    return T.Compose(transforms)

# use our dataset and defined transformations
dataset = NatureDataset('../shared/datasets/nature/train', get_transform(train=True))
dataset_test = NatureDataset('../shared/datasets/nature/test', get_transform(train=False))

import utils

# define training and validation data loaders
data_loader = torch.utils.data.DataLoader(
    dataset, batch_size=2, shuffle=True,
    collate_fn=utils.collate_fn)

data_loader_test = torch.utils.data.DataLoader(
    dataset_test, batch_size=1, shuffle=False,
    collate_fn=utils.collate_fn)
```

Now let's visualize one image from our dataset.

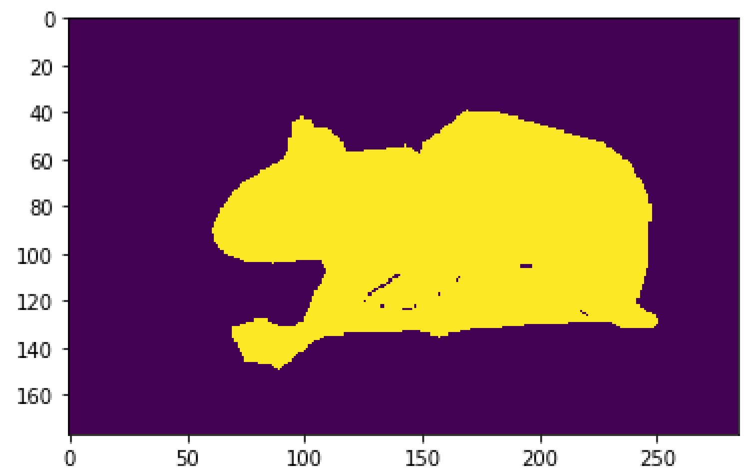
In [4]:

```python
import matplotlib.pyplot as plt

img, targets = dataset[506]

# np.transpose docs: https://numpy.org/doc/stable/reference/generated/numpy.transpose.html
# img is a PyTorch tensor, can convert it to a NumPy tensor by calling .numpy() on it.
plt.imshow(np.transpose(img.numpy(), (1, 2, 0)));
```

```
/opt/conda/envs/py37/lib/python3.7/site-packages/ipykernel_launcher.py:118: UserWarning: Creating a tensor fro
m a list of numpy.ndarrays is extremely slow. Please consider converting the list to a single numpy.ndarray wi
th numpy.array() before converting to a tensor. (Triggered internally at  ../torch/csrc/utils/tensor_new.cpp:2
01.)
```

And here's the corresponding mask.

In [5]:
```
plt.imshow(np.transpose(targets['masks'].numpy(), (1, 2, 0)), interpolation='none');
```



We'll be training the final layers of the pretrained Mask-RCNN model (with Resnet-50-FPN backbone) available in the `torchvision` package. So let's download the model:

In [6]:
```
model_p2 = torchvision.models.detection.maskrcnn_resnet50_fpn(pretrained=True)
```

Question 5

Recall what we said earlier: For training for the new dataset, we need to modify its box predictor (FastRCNNPredictor) and its mask predictor (MaskRCNNPredictor) to match with the new dataset. Complete the code cell below.

You may find these docs for `FastRCNNPredictor` and `MaskRCNNPredictor` useful:

```
In [154]: FastRCNNPredictor?
Init signature: FastRCNNPredictor(in_channels, num_classes)
Docstring:
Standard classification + bounding box regression layers
for Fast R-CNN.

Args:
    in_channels (int): number of input channels
    num_classes (int): number of output classes (including background)
Init docstring: Initializes internal Module state, shared by both nn.Module and ScriptModule.
File:           /usr/local/lib/python3.9/site-packages/torchvision/models/detection/faster_rcnn.py
Type:           type
Subclasses:
```

```
In [155]: MaskRCNNPredictor?
Init signature: MaskRCNNPredictor(in_channels, dim_reduced, num_classes)
Docstring:
A sequential container.
Modules will be added to it in the order they are passed in the constructor.
Alternatively, an ordered dict of modules can also be passed in.

To make it easier to understand, here is a small example::

    # Example of using Sequential
    model = nn.Sequential(
              nn.Conv2d(1,20,5),
              nn.ReLU(),
              nn.Conv2d(20,64,5),
              nn.ReLU()
            )

    # Example of using Sequential with OrderedDict
    model = nn.Sequential(OrderedDict([
              ('conv1', nn.Conv2d(1,20,5)),
              ('relu1', nn.ReLU()),
              ('conv2', nn.Conv2d(20,64,5)),
              ('relu2', nn.ReLU())
            ]))
Init docstring: Initializes internal Module state, shared by both nn.Module and ScriptModule.
File:           /usr/local/lib/python3.9/site-packages/torchvision/models/detection/mask_rcnn.py
Type:           type
Subclasses:
```

In [7]:
```
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torchvision.models.detection.mask_rcnn import MaskRCNNPredictor

# Our new dataset has 3 classes: butterfly, squirrel and background
num_classes = 3

# Get number of input features for the classifier
in_features = model_p2.roi_heads.box_predictor.cls_score.in_features

# TODO: replace the pre-trained head with a new one
model_p2.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

# Get number of input features for the mask predictor
in_features_mask = model_p2.roi_heads.mask_predictor.conv5_mask.in_channels
hidden_layer = 256

# TODO: replace the mask predictor with a new one
model_p2.roi_heads.mask_predictor = MaskRCNNPredictor(in_features_mask, hidden_layer, num_classes)
```

Training these layers can take several minutes on a CPU, so we've provided GPUs to make this faster. It should take ~5 mins to run the training in Part2. But before that, we want to ensure that PyTorch is able to access the GPU by printing the device PyTorch is currently (prints `cuda` if it's using a GPU, otherwise it prints `cpu`).

Now we want to freeze all the layers below these predictors. We can do this by setting the `.requires_grad` attribute of the parameters we want to freeze to `False` . Read more about `requires_grad` from [this PyTorch page on Autograd mechanics (https://pytorch.org/docs/stable/notes/autograd.html#excluding-subgraphs-from-backward)](https://pytorch.org/docs/stable/notes/autograd.html#excluding-subgraphs-from-backward).

In order to do the computation on a GPU, we have to move the model from main memory to GPU memory. This can be done by simply calling `.to(device)` on the model. See [the docs (https://pytorch.org/docs/stable/generated/torch.nn.Module.html#torch.nn.Module.to)](https://pytorch.org/docs/stable/generated/torch.nn.Module.html#torch.nn.Module.to) for more information.

In [8]:
```
import itertools

# Freeze model and move it to device
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
print("Using", device)

for param in model_p2.parameters():
    param.requires_grad = False

pred_params = itertools.chain(
    model_p2.roi_heads.mask_predictor.parameters(),
    model_p2.roi_heads.box_predictor.parameters()
)

for param in pred_params:
    param.requires_grad = True

model_p2 = model_p2.to(device)
```

```
Using cuda
```

We were able to verify that PyTorch is able to access a GPU. Now let's see the layers inside the `model_p2.roi_heads` to understand what we have modified here (we just modified `box_predictor` and `mask_predictor`). You could also verify the output below from figure 4 in the Mask-RCNN paper. You'll notice that it's the exact same network on the right part of that figure.

In [9]:
```
model_p2.roi_heads
```

Out[9]:
```
RoIHeads(
  (box_roi_pool): MultiScaleRoIAlign(featmap_names=['0', '1', '2', '3'], output_size=(7, 7), sampling_ratio=2)
  (box_head): TwoMLPHead(
    (fc6): Linear(in_features=12544, out_features=1024, bias=True)
    (fc7): Linear(in_features=1024, out_features=1024, bias=True)
  )
  (box_predictor): FastRCNNPredictor(
    (cls_score): Linear(in_features=1024, out_features=3, bias=True)
    (bbox_pred): Linear(in_features=1024, out_features=12, bias=True)
  )
  (mask_roi_pool): MultiScaleRoIAlign(featmap_names=['0', '1', '2', '3'], output_size=(14, 14), sampling_ratio
=2)
  (mask_head): MaskRCNNHeads(
    (mask_fcn1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu1): ReLU(inplace=True)
    (mask_fcn2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu2): ReLU(inplace=True)
    (mask_fcn3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu3): ReLU(inplace=True)
    (mask_fcn4): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu4): ReLU(inplace=True)
  )
  (mask_predictor): MaskRCNNPredictor(
    (conv5_mask): ConvTranspose2d(256, 256, kernel_size=(2, 2), stride=(2, 2))
    (relu): ReLU(inplace=True)
    (mask_fcn_logits): Conv2d(256, 3, kernel_size=(1, 1), stride=(1, 1))
  )
)
```

Question 6

We just printed the head architecture. From the above output, list all the layers we're training along their names.

For example, if we're training mask_fcn1 of mask_head you'll specify:

`mask_head.mask_fcn1` : 2d-Conv layer with 256 input channels, 256 output channels and kernel size = (2, 2).

Note: We're **only** asking for layers with trainable parameters.

Answer

`box_predictor.cls_score` : Linear layer with 1024 input features, 3 output features and bias=True

`box_predictor.bbox_pred` : Linear layer with 1024 input features, 12 output features and bias=True

`mask_predictor.mask_fcn_logits` : Conv2d layer with 256 input channels, 3 output channels, kernel_size=(1, 1), stride=(1, 1))

Both the dataloaders and model have been prepared for training. All that remains is to set an optimizer and a learning rate scheduler. When we create an optimizer, we have to provide it the list of trainable parameters.

In [10]:
```
# Declare optimizer and lr
params = [p for p in model_p2.parameters() if p.requires_grad]
optimizer = torch.optim.SGD(params, lr=0.005, momentum=0.9, weight_decay=0.0005)
lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                               step_size=3,
                                               gamma=0.1)
```

Question 7

How many trainable parameters are we passing to the optimizer?

Here's an example to calculate # of trainable parameters in a fully connected layer with:

1. an additive bias
2. in_channels = 1024
3. out_channels = 10

The number of trainable parameters here will be 1024*10 + 10 = 10250.

Answer

The number of trainable parameters are

$$(256 * 3) + (1024 * 3) + (1024 * 12) + (256 * 256) = 81664$$

Now we can finally start the training process.

In [11]:

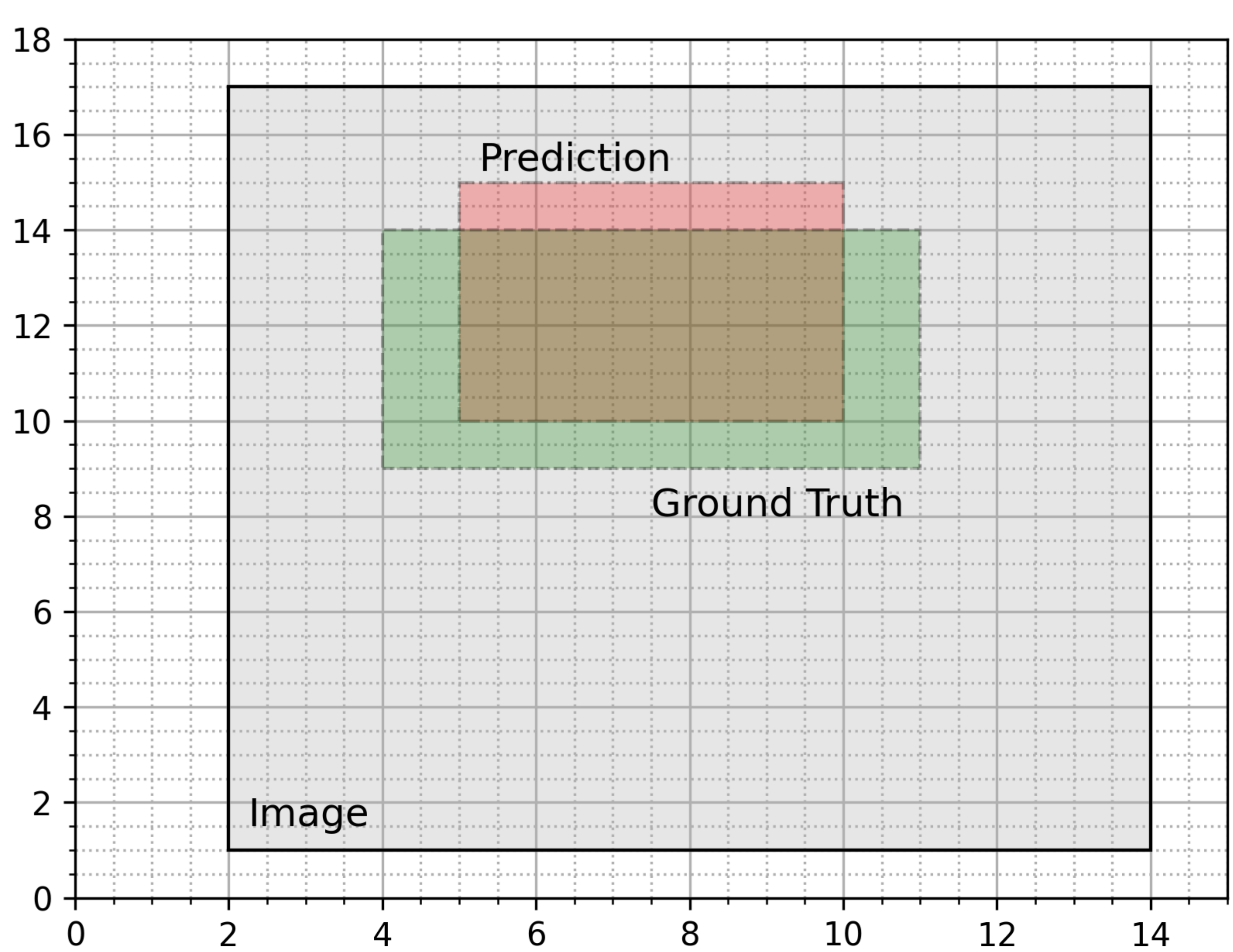
```python
num_epochs = 3

from engine import train_one_epoch, evaluate

for epoch in range(num_epochs):
    print("Epoch", epoch)
    # train for one epoch, printing every 10 iterations
    train_one_epoch(model_p2, optimizer, data_loader, device, epoch, print_freq=10)
    # update the learning rate
    lr_scheduler.step()
    # evaluate on the test dataset
    evaluate(model_p2, data_loader_test, device=device)
```

```
loss_box_reg: 0.0520 (0.0712)  loss_mask: 0.2051 (0.2570)  loss_objectness: 0.0056 (0.0057)  loss_rpn_box_re
g: 0.0111 (0.0136)  time: 0.2901  data: 0.0214  max mem: 1593
Epoch: [2]  [299/300]  eta: 0:00:00  lr: 0.005000  loss: 0.3260 (0.3880)  loss_classifier: 0.0359 (0.0417)
loss_box_reg: 0.0547 (0.0713)  loss_mask: 0.2018 (0.2559)  loss_objectness: 0.0036 (0.0057)  loss_rpn_box_re
g: 0.0088 (0.0134)  time: 0.2979  data: 0.0200  max mem: 1593
Epoch: [2] Total time: 0:01:28 (0.2952 s / it)
creating index...
index created!
Test:  [  0/200]  eta: 0:00:35  model_time: 0.1653 (0.1653)  evaluator_time: 0.0038 (0.0038)  time: 0.1765
data: 0.0071  max mem: 1593
Test:  [100/200]  eta: 0:00:13  model_time: 0.1321 (0.1238)  evaluator_time: 0.0042 (0.0041)  time: 0.1440
data: 0.0110  max mem: 1593
Test:  [199/200]  eta: 0:00:00  model_time: 0.1254 (0.1247)  evaluator_time: 0.0037 (0.0039)  time: 0.1390
data: 0.0103  max mem: 1593
Test: Total time: 0:00:27 (0.1392 s / it)
Averaged stats: model_time: 0.1254 (0.1247)  evaluator_time: 0.0037 (0.0039)
Accumulating evaluation results...
DONE (t=0.04s).
Accumulating evaluation results...
```

In order to analyze the output, you'll need to know what an IoU score is. You can read this blog: https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/ (https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/)

Question 8

In an image (grey box) of size 12x16, there is an object whose ground truth mask is the box with the dashed edge (green color) and the model's predicted mask is the box with the dash-dotted edge (red color). Calculate the IoU score for this prediction.



Answer

$$IoU = \frac{Insersection}{Union} = \frac{5*4}{(5*7)+(1*5)} = \frac{20}{40} = 0.5$$

Let's see the model's prediction for a sample from the test set.

In [12]:
```
img, target = dataset_test[1]

# put the model in evaluation mode
model_p2.eval()

with torch.no_grad():
    prediction = model_p2([img.to(device)])
```

Here's the sample image.

In [13]:
```
Image.fromarray(img.mul(255).permute(1, 2, 0).byte().numpy())
```

Out[13]:

And here's the mask generated by the finetuned model.

In [14]:
```
Image.fromarray(prediction[0]['masks'][0, 0].mul(255).byte().cpu().numpy())
```

Out[14]:

Part 3: Finetuning the Entire Network

Let's see what happens when we fine-tune the entire network. That is, instead of just learning the weights in the final layers, we'll let the weights of the entire network change during the training process.

In [15]:
```
model_p3 = torchvision.models.detection.maskrcnn_resnet50_fpn(pretrained=True)
```

Question 9 ¶

Just like in question 5, we're interested in fine-tuning the pretrained model for our new dataset, so we will again need to modify its box predictor (FastRCNNPredictor) and its mask predictor (MaskRCNNPredictor) to match with our new dataset. Complete the code cell below. Hint: This is exactly the same as question 5.

In [16]:
```
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torchvision.models.detection.mask_rcnn import MaskRCNNPredictor

# Our new dataset has 3 classes: butterfly, squirrel and background
num_classes = 3

# Get number of input features for the classifier
in_features = model_p3.roi_heads.box_predictor.cls_score.in_features

# TODO: replace the pre-trained head with a new one
model_p3.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

# Get number of input features for the mask predictor
in_features_mask = model_p3.roi_heads.mask_predictor.conv5_mask.in_channels
hidden_layer = 256

# TODO: replace the mask predictor with a new one
model_p3.roi_heads.mask_predictor = MaskRCNNPredictor(in_features_mask, hidden_layer, num_classes)
```

Again, let's ensure that we're using the GPU by printing the device info. This finetuning process takes even longer because we have more trainable parameters, hence there will be more computations.

In [17]:
```
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
print("Using", device)

model_p3 = model_p3.to(device)
```

```
Using cuda
```

We'll use the same optimizer and learning rate scheduler as before.

In [18]:
```
params = [p for p in model_p3.parameters() if p.requires_grad]
optimizer = torch.optim.SGD(params, lr=0.005, momentum=0.9, weight_decay=0.0005)
lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                               step_size=3,
                                               gamma=0.1)
```

Let's start the finetuning process.

In [19]:
```
num_epochs = 3

from engine import train_one_epoch, evaluate

for epoch in range(num_epochs):
    print("Epoch", epoch)
    # train for one epoch, printing every 10 iterations
    train_one_epoch(model_p3, optimizer, data_loader, device, epoch, print_freq=10)
    # update the learning rate
    lr_scheduler.step()
    # evaluate on the test dataset
    evaluate(model_p3, data_loader_test, device=device)
```

```
data: 0.0102  max mem: 4593
Test: Total time: 0:00:27 (0.1361 s / it)
Averaged stats: model_time: 0.1249 (0.1225)  evaluator_time: 0.0034 (0.0030)
Accumulating evaluation results...
DONE (t=0.04s).
Accumulating evaluation results...
DONE (t=0.04s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.641
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.960
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.761
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.589
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.658
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.648
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.722
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.722
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.658
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.736
```

We'll generate the output for the same image, but this time with the new finetuned model.

In [20]:
```
img, target = dataset_test[1]
type(img)
# put the model in evaluation mode
model_p3.eval()
with torch.no_grad():
    prediction = model_p3([img.to(device)])

Image.fromarray(img.mul(255).permute(1, 2, 0).byte().numpy())
```

Out[20]:

Here's the predicted image.

In [21]:
```
Image.fromarray(prediction[0]['masks'][0, 0].mul(255).byte().cpu().numpy())
```

Out[21]:

Question 10

Does this model perform better than the trained model in part2? Explain why.

Answer

The mAP for the model in part2 is 53.7% and 58.9% for bounding box and mask, respectively. Whereas, in the last fine-tuning, we get mAP for the model as 64.1% and 73.5% for bbox and mask, respectively.

Thus, it can be said that this model performs better than the previous model.

Part 4: Extra Credit Questions

Question 11 [15 points]

Can fully convolutional networks (FCN) be used for object detection? In Mask-RCNN we have 3 branches — mask, classification, and bounding box regression — out of which the last 2 have fully connected (FC) layers. Can this entire pipeline be replaced by a fully convolutional network? If possible, give 1 or 2 networks to support your claim.

Answer

Yes, there are several studies that show that FCN can be used for end-to-end object detection.

`Paper 1:` "Object Detection by R-FCN: This is a region-based fully convolutional object detector. To achieve this goal, the authors proposes a position-sensitive score maps to address a dilemma between translation-invariance in image classification and translation-variance in object detection. Their method can thus naturally adopt fully convolutional image classifier backbones, such as the latest Residual Networks (ResNets) for object detection. They showed competitive results on the PASCAL VOC datasets with the 101-layer ResNet.

`Link to above paper:` [https://arxiv.org/abs/2012.03544 (https://arxiv.org/abs/2012.03544)](https://arxiv.org/abs/2012.03544)

`Paper 2:` The authors adapted the pretrained networks (AlexNet, the VGG net, and GoogLeNet) into fully convolutional networks and transfer their learned representations by fine-tuning to the segmentation task. They, then defined a skip architecture that combines semantic information from a deep, coarse layer with appearance information from a shallow, fine layer to produce accurate and detailed segmentations. Their fully convolutional networks achieve improved segmentation of PASCAL VOC (30% relative improvement to 67.2% mean IU on 2012), NYUDv2, SIFT Flow, and PASCAL-Context, while inference took one tenth of a second for a typical image.

`Link to above paper:` E. Shelhamer, J. Long and T. Darrell, "Fully Convolutional Networks for Semantic Segmentation," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 39, no. 4, pp. 640-651, 1 April 2017, doi: 10.1109/TPAMI.2016.2572683.

Question 12 [5 points]

What is the advantage of using CONV layers over FC/Dense layers?

Answer

A mask encodes an input object's spatial layout. Thus, unlike class labels or box offsets that are collapsed into short output vectors by fully-connected (fc) layers, extracting the spatial structure of masks can be addressed naturally by the pixel-to-pixel correspondence provided by convolutions.

Specifically, we predict the mask from each RoI using an FCN (Fully Conv Layers). This allows each layer in the mask branch to maintain the object spatial layout explicitly without collapsing it into a vector representation that lacks spatial dimensions.

Another advantage is, unlike previous methods that resort to fc layers for mask prediction, the fully convolutional representation requires fewer parameters, and is more accurate as demonstrated by experiments.

Vision meets Machine Learning HW3, Part 2 ¶

Continuing from HW2 Part 2, We will train a face segmentation model using the mask generation function

In [1]:

```
import torch
import torchvision
import torch
import torchvision
import torch.nn as nn
import cv2
from collections import defaultdict
%matplotlib inline
import numpy as np
from matplotlib import pyplot as plt
```

```
/opt/conda/envs/py37/lib/python3.7/site-packages/tqdm/auto.py:22: TqdmWarning: IProgress not found. Please upd
ate jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html (https://ipywidg
ets.readthedocs.io/en/stable/user_install.html)
  from .autonotebook import tqdm as notebook_tqdm
Unable to revert mtime: /opt/conda/fonts
```

In [2]:

```
mom = 0.9          # momemtum in batch normalization
lr = 0.0001        # learning rate in model training
i_up_lim = 10      # the number of epochs
```

Datagen

We will be using the 3500 images in shared/datasets/smaller_faces

Generate masks for the first 3500 images from the smaller_faces.txt

Update the paths in smaller_faces.txt to the correct path (should point to the smaller_faces folder in shared/datasets)

In [3]:

```
import cv2
import numpy as np
import math


#todo reuse hw2 code for datagen(Ellipse), This could take around 30-45 mins. make you use imwrite to save c

%run Mask.py
```

Custom dataset

Create a custom dataset for this problem as we did in part1 of this HW

In [4]:
```
import os
import torch
import numpy as np
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, utils
import glob
from PIL import Image

class face_segment(Dataset):


    def __init__(self, x, y, x_transform = None, y_transform = None):
        #todo
        imgs = glob.glob(x + '/*.jpg')
        self.imgs = sorted(imgs)
        #print("The sorted imagelist is ", self.imgs)

        masks = glob.glob(y + '/*.jpg')
        self.masks = sorted(masks)
        #print("The sorted maskslist is ", self.masks)

        self.x_transform = x_transform
        self.y_transform = y_transform

    def __len__(self):
        #todo
        return len(self.imgs)

    def __getitem__(self, idx):
        #todo
        #No fancy transforms needed. only the ones defined in the cell below.
        #make sure the your masks are 1 channel and it is populated only with 0s or 1s

        img_path = self.imgs[idx]
        mask_path = self.masks[idx]

        img = Image.open(img_path).convert("RGB")
        mask = np.array(Image.open(mask_path))*255

        if self.x_transform is not None:
            img = self.x_transform(img)
        if self.y_transform is not None:
            mask = self.y_transform(mask)

        return img, mask
```

In [5]:
```
import torchvision.transforms as T
def get_transform(X):
    transforms = []
    transforms.append(T.ToTensor())
    if X:
        transforms.append(T.Resize((800,800), interpolation=T.InterpolationMode.BILINEAR) )
    else:
        transforms.append(T.Resize((200,200), interpolation=T.InterpolationMode.NEAREST) )

    return T.Compose(transforms)

```

Dataset and Dataloader

In [6]:
```
# use our dataset and defined transformations, it would look something like vv
# dataset = face_segment(X, Y, get_transform(train=True), get_transform(train=False))

X = "/home/jovyan/shared/datasets/smaller_faces"
Y = "/home/jovyan/HW3/Masks"
dataset = face_segment(X, Y, get_transform(True), get_transform(False))

# define training data loaders, it would look something like vv
# data_loader = torch.utils.data.DataLoader(dataset, batch_size=4, shuffle=True)

data_loader = torch.utils.data.DataLoader(dataset, batch_size=4, shuffle=True)

# idx=3000
# img, mask = dataset[idx]
# plt.imshow(img[0])
# #print(mask[0])
```

In [7]:
```
# plt.imshow(mask[0])
# print(mask.shape)
# print(img.shape)
# #print(img[0])
```

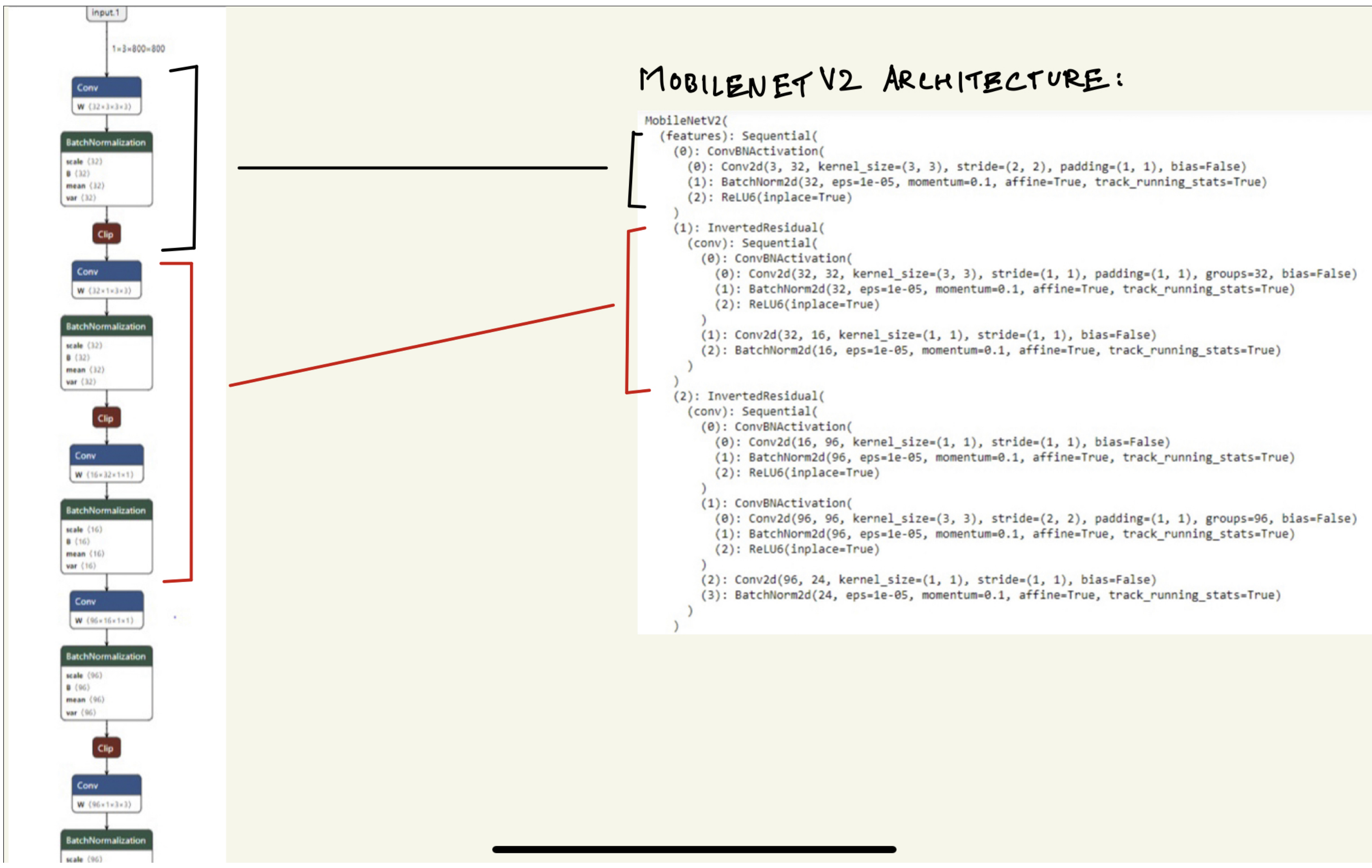
Centerface model

Hints: A lot of the code you need has already been put in comments below. please go through them and come back to this description for more clarity.

We will now be reconstructing a modified version of a model (modded_centerface.onnx) described in [https://arxiv.org/abs/1911.03599 (https://arxiv.org/abs/1911.03599)](https://arxiv.org/abs/1911.03599). Instead of building the entire detection pipeline, we will only create the classification branch (segmentation). This model uses mobilenet_v2 as its backbone (encoder) and instead of coding the entire mobilenet_v2 from scratch we will use mobilenet_v2 from torch.hub.

Looking at the netron graph, you will realise that we need certain intermediate outputs of the mobilenet_v2 model. How to get those outputs are described in the comments in the forward function.

You have to look at the at the outermost block numbers to get the value of which layer's output is added to which lane in the final block of the netron model as shown in this image. You can print the mobilenet_v2 model and compare it with the netron graph as done below.



In [8]:

```
import torch.nn as nn
from torch.nn import Sequential
import torch
import numpy as np
from torchvision import models
import torch.nn.functional as f
import cv2 as cv
from torchsummary import summary

class centerface(nn.Module):
    def __init__(self):
        super(centerface, self).__init__()
#         These are all the layers you will need. You may use these as is or make the model yourself from s
#         You have to fill out the forward function according to the netron graph

        mobilenet_v2 = torch.hub.load('pytorch/vision:v0.10.0', 'mobilenet_v2', pretrained=True)
        self.mobilenet_v2=nn.ModuleList(list(mobilenet_v2.features)[:-1])

        self.mobilenet_v2_final = Conv2d(320,24,1,1)
        self.lane_0_transpose = ConvTranspose2d(24,24,2,2)
        self.lane_1_conv = Conv2d(96,24,1,1)
        self.lane_0_1_transpose =  ConvTranspose2d(24,24,2,2)
        self.lane_2_conv = Conv2d(32,24,1,1)
        self.lane_0_1_2_transpose = ConvTranspose2d(24,24,2,2)
        self.lane_3_conv = Conv2d(24,24,1,1)
        self.final_merge = Conv2d(24,24,3,1,same_padding=True)
        self.hmap = nn.Conv2d(24, 1, 1, 1, padding=0,bias=True)
        self.sig_out=nn.Sigmoid()
        self._initialize_weights()

#         result = []
#         for idx, model in enumerate(self.mobilenet_v2):
#             #print("idx  is ", idx)
#             #print("model ", model)
#             if idx in [3,6,13,17]:
#                 result.append(model)
#         print ("The result is as follows ")
#         print(result)

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.normal_(m.weight, std=0.01)
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    def forward(self, x):

        #todo
#        You will need to get intermediate outputs from the mobilenet_v2 backbone. You can do that by itera
#        over the nn.ModuleList(list(mobilenet_v2.features)[:-1]) we created previously.
#        ModuleList can be indexed like a regular Python list, but modules it contains are properly registe
#        and will be visible by all Module methods.
# E.g,
#         result=[]
#         for idx,model in enumerate(self.mobilenet_v2):
#             x=model(x)
#             if(idx in {whichever layer indices you need (You will get this information from the netron gr
#                 result.append(x)

        result = []
        for idx, model in enumerate(self.mobilenet_v2):
            x =model(x)
            if idx in [3,6,13,17]:
                result.append(x)

        x_lane_0 = result[3]
        x_lane_0 = self.mobilenet_v2_final(x_lane_0)
        x_lane_0 = self.lane_0_transpose(x_lane_0)

        x_lane_1 = result[2]
        x_lane_1 = self.lane_1_conv(x_lane_1)

        x1 = x_lane_0 + x_lane_1
        x1 = self.lane_0_1_transpose(x1)

        x_lane_2 = result[1]
        x_lane_2 = self.lane_2_conv(x_lane_2)

        x2 = x1 + x_lane_2
        x2 = self.lane_0_1_2_transpose(x2)

        x_lane_3 = result[0]
        x3 = self.lane_3_conv(x_lane_3)

        x_final = x2 + x3
        x_final = self.final_merge(x_final)
```

```
        x_final = self.hmap(x_final)
        x = self.sig_out(x_final)

        return x


class Conv2d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride=1, relu=True, same_padding=False, bn=
        super(Conv2d, self).__init__()
        padding = int((kernel_size - 1) / 2) if same_padding else 0
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding=padding,bias=False)
        self.bn = nn.BatchNorm2d(out_channels, eps=0.001, momentum=mom, affine=True,track_running_stats=Fal
        self.relu = nn.ReLU(inplace=False) if relu else None

    def forward(self, x):
        x = self.conv(x)
        if self.bn is not None:
            x = self.bn(x)
        if self.relu is not None:
            x = self.relu(x)
        return x

class ConvTranspose2d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=2, stride=2, relu=True, same_padding=False, b
        super(ConvTranspose2d, self).__init__()
        padding = int((kernel_size - 1) / 2) if same_padding else 0
        self.conv = nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride, padding=padding,bias
        self.bn = nn.BatchNorm2d(out_channels, eps=0.001, momentum=mom, affine=True,track_running_stats=Fal
        self.relu = nn.ReLU(inplace=False) if relu else None

    def forward(self, x):
        x = self.conv(x)
        if self.bn is not None:
            x = self.bn(x)
        if self.relu is not None:
            x = self.relu(x)
        return x

#k = centerface()
#inputs = torch.randn(1,3,800,800,requires_grad=True)
#torch_out = k(inputs)
#summary(k,(3,200,200))
#torch.onnx.export(k,inputs,"My_CNN_model.onnx", export_params=True)
```

Boilerplate utilities

In [9]:

```
import time
class AverageMeter(object):
    """Computes and stores the average and current value"""
    def __init__(self):
        self.reset()

    def reset(self):
        self.val = 0
        self.avg = 0
        self.sum = 0
        self.count = 0

    def update(self, val, n=1):
        self.val = val
        self.sum += val * n
        self.count += n
        self.avg = self.sum / self.count

def log_print(text, color=None, on_color=None, attrs=None):
    if cprint is not None:
        cprint(text, color=color, on_color=on_color, attrs=attrs)
    else:
        print(text)
try:
    from termcolor import cprint
except ImportError:
    cprint = None
```

Initialize model

In [10]:
```
#todo
model = centerface()
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
model = centerface().to(device).float()
```

```
Downloading: "https://github.com/pytorch/vision/archive/v0.10.0.zip" to /home/jovyan/.cache/torch/hub/v0.10.0.
zip
Downloading: "https://download.pytorch.org/models/mobilenet_v2-b0353104.pth" to /home/jovyan/.cache/torch/hub/
checkpoints/mobilenet_v2-b0353104.pth
100%|██████████| 13.6M/13.6M [00:00<00:00, 68.3MB/s]
Using cache found in /home/jovyan/.cache/torch/hub/pytorch_vision_v0.10.0
```

Training loop

You can take a look at this link to get an understanding of the training function.

[https://pytorch.org/tutorials/beginner/introyt/trainingyt.html (https://pytorch.org/tutorials/beginner/introyt/trainingyt.html)](https://pytorch.org/tutorials/beginner/introyt/trainingyt.html)

In [11]:
```
def train(data_loader, model, criterion, optimizer, epoch):
    losses = AverageMeter()
    batch_time = AverageMeter()
    data_time = AverageMeter()
    train_loader = data_loader

    log_text = 'epoch %d, processed %d samples, lr %.10f' % (epoch, epoch * len(train_loader.dataset), lr)
    # this is just for logs
    log_print(log_text, color='green', attrs=['bold']) # this is just for logs

    # todo
    end = time.time()

    # This is pseudocode.
    for i, data in enumerate(train_loader):
        imgs, masks = data
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        imgs, masks = imgs.to(device), masks.to(device)

        #I am filling up the boiler plate utilities just for consistent output logs.
        data_time.update(time.time() - end) # this is just for logs
        #todo...
        #....

        optimizer.zero_grad()
        outputs = model(imgs)
        loss = criterion(outputs, masks)
        losses.update(float(loss), 1)  # this is just for logs
        loss.backward()
        optimizer.step()

        batch_time.update(time.time() - end)
        end = time.time()

        if i % 20 == 0:
            log_text = (('Epoch: [{0}][{1}/{2}]\t'
                    'Patch {patch_num:d}\t'
                    'Time {batch_time.val:.3f} ({batch_time.avg:.3f})\t'
                    'Data {data_time.val:.3f} ({data_time.avg:.3f})\t'
                    'Loss {loss.val:.8f} ({loss.avg:.4f})\t'
                    ).format(epoch, i, len(train_loader), patch_num=0, batch_time=batch_time,
                    data_time=data_time, loss=losses))

            log_print(log_text, color='green', attrs=['bold'])


```

Train the model

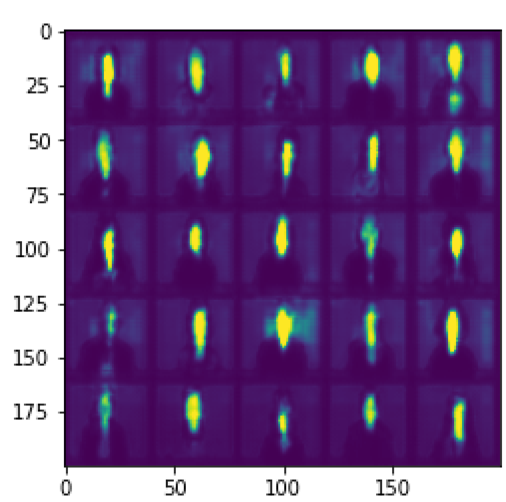
In [12]:
```
optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr,weight_decay=5e-3)
criterion = nn.BCELoss(reduction='mean').cuda()

for i in range(i_up_lim):
    train(data_loader, model, criterion, optimizer, i)
    torch.save(model, 'model.pt')
```

```
Epoch: [9][520/875]     Patch 0 Time 0.344 (0.339)      Data 0.149 (0.144)      Loss 0.06680539 (0.1007)
Epoch: [9][540/875]     Patch 0 Time 0.341 (0.339)      Data 0.145 (0.144)      Loss 0.08494115 (0.1004)
Epoch: [9][560/875]     Patch 0 Time 0.346 (0.339)      Data 0.153 (0.144)      Loss 0.08154360 (0.1007)
Epoch: [9][580/875]     Patch 0 Time 0.334 (0.339)      Data 0.132 (0.144)      Loss 0.10970800 (0.1005)
Epoch: [9][600/875]     Patch 0 Time 0.341 (0.339)      Data 0.146 (0.144)      Loss 0.30756873 (0.1013)
Epoch: [9][620/875]     Patch 0 Time 0.333 (0.339)      Data 0.138 (0.144)      Loss 0.09629402 (0.1011)
Epoch: [9][640/875]     Patch 0 Time 0.331 (0.339)      Data 0.131 (0.144)      Loss 0.06516825 (0.1012)
Epoch: [9][660/875]     Patch 0 Time 0.316 (0.339)      Data 0.122 (0.144)      Loss 0.05113391 (0.1013)
Epoch: [9][680/875]     Patch 0 Time 0.326 (0.339)      Data 0.131 (0.144)      Loss 0.15673082 (0.1015)
Epoch: [9][700/875]     Patch 0 Time 0.344 (0.339)      Data 0.149 (0.144)      Loss 0.07833005 (0.1017)
Epoch: [9][720/875]     Patch 0 Time 0.338 (0.339)      Data 0.142 (0.144)      Loss 0.06947359 (0.1017)
Epoch: [9][740/875]     Patch 0 Time 0.409 (0.339)      Data 0.215 (0.144)      Loss 0.06252707 (0.1015)
Epoch: [9][760/875]     Patch 0 Time 0.337 (0.339)      Data 0.141 (0.144)      Loss 0.10605373 (0.1019)
Epoch: [9][780/875]     Patch 0 Time 0.347 (0.339)      Data 0.154 (0.144)      Loss 0.13296019 (0.1026)
Epoch: [9][800/875]     Patch 0 Time 0.333 (0.339)      Data 0.140 (0.144)      Loss 0.06059437 (0.1028)
Epoch: [9][820/875]     Patch 0 Time 0.329 (0.339)      Data 0.131 (0.144)      Loss 0.04567843 (0.1033)
Epoch: [9][840/875]     Patch 0 Time 0.354 (0.339)      Data 0.160 (0.144)      Loss 0.17501679 (0.1034)
Epoch: [9][860/875]     Patch 0 Time 0.356 (0.339)      Data 0.161 (0.144)      Loss 0.08343604 (0.1030)
```

Test the model and visualise the predictions

Pass the test image (test_grid.jpg) to your model and visualise the model's prediction.

In [17]:
```
model.eval()

img = Image.open("/home/jovyan/HW3/test_grid.jpg")

transform = transforms.Compose([transforms.Resize((800,800)),transforms.ToTensor()])
input_img = transform(img)
input_img = input_img.unsqueeze(0)

if torch.cuda.is_available():
    input_img = input_img.to('cuda')
    model.to('cuda')

with torch.no_grad():
    prediction = model(input_img)
prediction = prediction.detach().cpu()
prediction = np.squeeze(prediction)
plt.imshow(prediction)
plt.show()
```



In [ ]:
```

```

In [1]:

```
def Mask():
    import cv2
    import numpy as np
    from IPython.display import Image
    from PIL import Image as convert_to_image
    import matplotlib.pyplot as plt
    import os

    replace_path = "/home/jovyan/shared/datasets/smaller_faces/"
    source_path = "/home/jovyan/HW3/smaller_faces.txt"

    with open(source_path, 'r') as file :
        filedata = file.read()

    filedata = filedata.replace("smaller_faces/", replace_path)

    with open('smaller_faces_dup.txt', 'w') as file:
        file.write(filedata)

    ### -----------------------------------------------------------------------------------------------#

    def create_folder():
        path = "/home/jovyan/HW3/Masks/"
        isExist = os.path.exists(path)

        if not isExist:
            os.makedirs(path)
        return path

    ### -----------------------------------------------------------------------------------------------#

    def ellipse(box_coord, image_width, image_height, w_x, h_y):
        image = np.zeros((image_height, image_width), dtype=int)
        center = [box_coord[0]+w_x//2, box_coord[1]+h_y//2]
        x, y = center[1], center[0]  # Rows and columns of the center

        cx,cy = [x, y]
        queue = [[cx,cy]]
        neighbors = [[1, 0], [-1, 0], [0, 1], [0, -1]]

        if w_x < h_y:
            min_ax = w_x//2
            maj_ax = h_y//2
        else:
            min_ax = h_y//2
            maj_ax = w_x//2

        def dist(a,b):
            return np.sqrt((((a[0]-b[0])**2)*(min_ax**2)) + (((a[1]-b[1])**2)*(maj_ax**2)))

        while(len(queue)>0):
            tx, ty = queue.pop(0)
            if(dist([cx,cy], [tx,ty])) >= (min_ax*maj_ax) or image[ty,tx]==1:
                continue  # Do not need to fill this value, because outside the bounds
            image[ty,tx] = 1 # mark the pixel; we'll multiply the mask by 255 for visualization later

            for i,j in neighbors:
                x = tx + i
                y = ty + j
                if(0<=x<img.shape[0] and 0<=y<img.shape[1]): #boundary check to make sure co-ordinates are
                    queue.append((x,y))

        return np.transpose(image)

    ### -----------------------------------------------------------------------------------------------#

    def lookup():
        heading_list = []
        n_faces = []
        dim_list = []
        heading_name = []

        with open("smaller_faces_dup.txt") as file:
            lines = file.readlines()
        file.close()

        lookup = "/shared/datasets/smaller_faces/"

        for num, line in enumerate(lines, 0):
            if lookup in line:
                heading_name.append(line.strip())
                heading_list.append(int(num))
                n = int(lines[num+1])
                n_faces.append(n)
                for i in range(n):
                    dim_list.append(list(map(int, lines[num+2+i].split()[0:4])))

        return heading_list, n_faces, dim_list, heading_name

    ### -----------------------------------------------------------------------------------------------#
```

```python

    def loops(dim_list, image_width, image_height):
        box_coord = [dim_list[0], dim_list[1]]      # The coordinates of the bounding box [x-dir, y-dir]
        w_x = dim_list[2]   # Width of the bouning box
        h_y = dim_list[3]   # Height of the bounding box

        image_ellipse = ellipse(box_coord, image_width, image_height, w_x, h_y)

        return image_ellipse

    ### ----------------------------------------------------------------------------------------------------#

    def print_to_file(image, image_width, image_height,  heading_name, img):
        path = create_folder()
        print(path)
        image_ = np.zeros((image_width,image_height,1),dtype= np.uint8)
        image_[:,:,0] = image


        img_save = heading_name[42:]
        #dst = cv2.addWeighted(original_image, 0.5, image_*255, 0.5, 0.0)
        save_path = "/home/jovyan/HW3/Masks"+img_save
        cv2.imwrite(save_path, image_)
        # plt.imshow(image_)

    ### ----------------------------------------------------------------------------------------------------#

    from itertools import islice

    heading_list, n_faces, dim_list, heading_name = lookup()

    temp = iter(dim_list)
    dim_list_grouped = [list(islice(temp, 0, i)) for i in n_faces]

    for i in range(len(heading_name)):
        path = heading_name[i]
        img = cv2.imread(path)
        image_width = img.shape[0]
        image_height = img.shape[1]
        image_ellipse = np.zeros((image_width,image_height), dtype = int)

        for j in range(n_faces[i]):
            image_ellipse += loops(dim_list_grouped[i][j], image_width, image_height)

        print_to_file(image_ellipse, image_width, image_height, heading_name[i], img)

    return

Mask()
```

```
/home/jovyan/HW3/Masks/
/home/jovyan/HW3/Masks/
/home/jovyan/HW3/Masks/
/home/jovyan/HW3/Masks/
/home/jovyan/HW3/Masks/
/home/jovyan/HW3/Masks/
/home/jovyan/HW3/Masks/
/home/jovyan/HW3/Masks/
/home/jovyan/HW3/Masks/
/home/jovyan/HW3/Masks/
/home/jovyan/HW3/Masks/
/home/jovyan/HW3/Masks/
/home/jovyan/HW3/Masks/
/home/jovyan/HW3/Masks/
/home/jovyan/HW3/Masks/
/home/jovyan/HW3/Masks/
/home/jovyan/HW3/Masks/
/home/jovyan/HW3/Masks/
/home/jovyan/HW3/Masks/
```