In GOD WE TRUST

Deep Learning Course

2021-2022 Fall Semester

Final Project

Object Detection + Depth Estimation Network

Ali Arasteh
(400206154)
Omid Sharafi
(400201518)

Instructor:
Dr. Emad Fatemizadeh

February 4, 2022

Sharif
University
of
Technology

Department of Electrical Engineering

Contents

1 Preparing Dataset

The input data-set format is .mat. We usually use the Scipy.loadmat library for opening .mat files. Since the version of this file is Matlab v7.3, we had to use publicly available library mat73.

Preparing Dataset

```
[ ] if not os.path.exists('/content/nyu_depth_v2_labeled.mat'):
      !wget http://horatio.cs.nyu.edu/mit/silberman/nyu_depth_v2/nyu_depth_v2_labeled.mat

[ ] !pip install mat73
    import mat73
    data_dict = mat73.loadmat('/content/nyu_depth_v2_labeled.mat')
    images = data_dict['images']
    depths = data_dict['depths']
    labels = data_dict['labels']
    names = data_dict['names']

[ ] images = np.moveaxis(images, -1, 0)
    depths = np.moveaxis(depths, -1, 0)
    labels = np.moveaxis(labels, -1, 0)

[ ] with open('/content/drive/MyDrive/DL - Project/images.pkl', 'wb') as f:
      pickle.dump(images, f)
    with open('/content/drive/MyDrive/DL - Project/depths.pkl', 'wb') as f:
      pickle.dump(depths, f)
    with open('/content/drive/MyDrive/DL - Project/labels.pkl', 'wb') as f:
      pickle.dump(labels, f)
    with open('/content/drive/MyDrive/DL - Project/names.pkl', 'wb') as f:
      pickle.dump(names, f)
```

The above library occupied a considerable amount of the colab RAM. So, after loading and converting the data-set to NumPy format, we re-saved images, depths, labels, and names of all 1449 images using the Pickle library. Also, for loading images from the disk during the network training process, we saved all of them separately in the NumPy format.

```
for i in range(len(images)):
  with open('/content/drive/MyDrive/DL - Project/images/'+str(i)+'.npy', 'wb') as f:
    np.save(f, images[i])
for i in range(len(depths)):
  with open('/content/drive/MyDrive/DL - Project/depths/'+str(i)+'.npy', 'wb') as f:
    np.save(f, depths[i])
for i in range(len(labels)):
  with open('/content/drive/MyDrive/DL - Project/labels/'+str(i)+'.npy', 'wb') as f:
    np.save(f, labels[i])
```

A sample of the data-set is available below.

Figure 1: Image and its depth

For training the object detector network, we needed to extract the objects' bounding boxes and labels from the data-set masks. In this regard, we developed a function (Image-boxer) that would give our desired output from the given image mask. We ran this function over all the data-set images and saved the results for further use.

This function searches all the image pixels and makes connected regions for each label based on the connectivity of same-labeled pixels. Then, it will detect the borders of each bounding box using positions of included pixels. In the end, it will remove small objects according to their size and combine overlapped bounding boxes with the non-maximum suppression method. [8]

Figure 2: The image bounding boxes and the refrigerator mask (we can set the size threshold and region of view of each pixel)

2 Object Detection

For doing the object detection task, first, we tried the Faster RCNN Model of PyTorch. We did not get an appropriate result, so we switched to the YOLO-v3 object detector. We will explain each method and its results in the following parts.

2.1 Faster RCNN Model

The faster RCNN Model is an object detector available in torchvision. Models. We first loaded pre-trained implementation of it from the torchvision library. This model was trained on 1280 classes of outdoor objects. So the original network performance on our data-set images, which are indoor scenes, was weak.

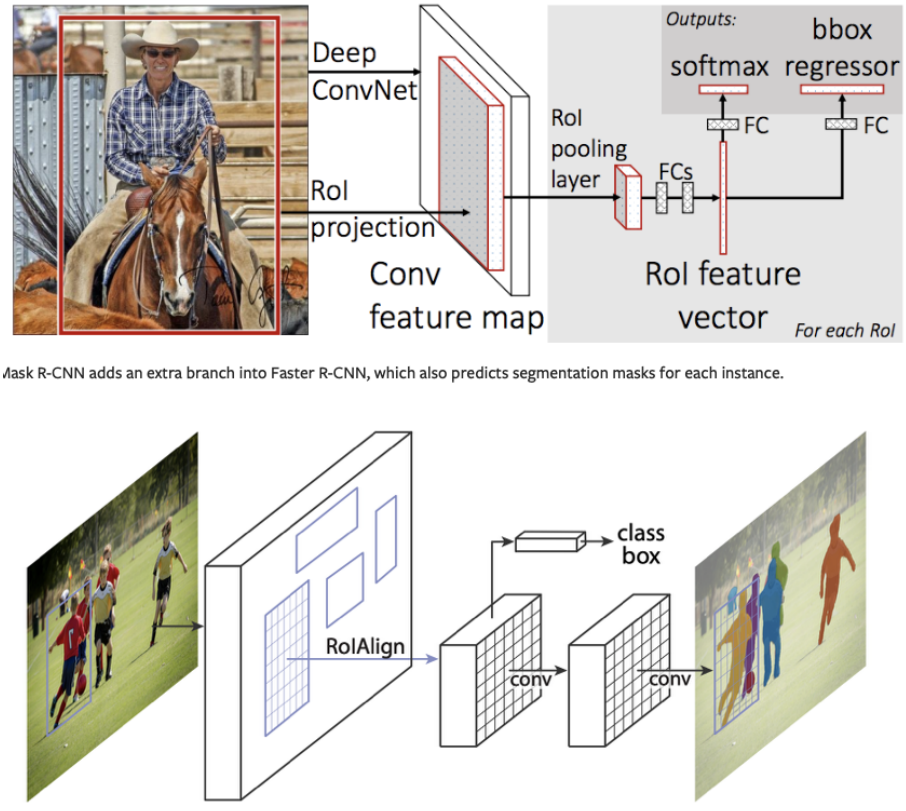

Figure 3: Faster RCNN Model [2]

For fine-tuning the network [9], we loaded the bounding boxes and labels of each data-set image provided from the previous part. We also updated the number of the network classes according to our data set, which has 894 + 1 (background) labels.

```
import torchvision
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)
num_classes = len(names)+1
in_features = model.roi_heads.box_predictor.cls_score.in_features
model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)
model.to(device)

torch.save(model.state_dict(), '/content/drive/MyDrive/DL - Project/model')

model.load_state_dict(torch.load('/content/drive/MyDrive/DL - Project/model'))

import cv2
lr = 0.001
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
p = np.ones(1449)

model.train()
```

Figure 4: Fine Tuning Fast RCNN Model

Then we trained the model for about 40 epochs in sample mode. Different scenarios we tried resulted in poor performance (network converged to the point that it did propose no region).

```
for i in range(len(boxes)):
  new_boxes.append([(boxes[i, 0]-y)*1.25, (boxes[i, 1]-x)*1.25, (boxes[i, 2]-y)*1.25, (boxes[i, 3]-x)*1.25])
  new_labels.append(box_labels[i])
if len(new_boxes) == 0:
  continue
new_boxes = np.array(new_boxes).astype(np.int64)
new_labels = new_labels
img = np.moveaxis(img, -1, 0)
masks = np.zeros((len(new_boxes), 480, 640))
for index in range(len(new_boxes)):
  for x in range(480):
    for y in range(640):
      masks[index, x, y] = (x>=new_boxes[index][0] and x<=new_boxes[index][2] and y>=new_boxes[index][1] and y<=new_boxes[index][3] and label[x, y]==new_labels[index])
target = {}
target["boxes"] = torch.as_tensor(new_boxes, dtype=torch.float32).to(device)
target["labels"] = torch.as_tensor(new_labels, dtype=torch.int64).to(device)
target["masks"] = torch.as_tensor(masks, dtype=torch.uint8).to(device)
target["image_id"] = torch.as_tensor(image_number, dtype=torch.float32).to(device)
target["area"] = torch.as_tensor((new_boxes[:, 3] - new_boxes[:, 1]) * (new_boxes[:, 2] - new_boxes[:, 0]), dtype=torch.float32).to(device)
target["iscrowd"] = torch.zeros((len(new_boxes),), dtype=torch.int64).to(device)
for i in range(10):
  model_in = torch.Tensor(img[np.newaxis, ...])
  model_in = model_in.to(device)
  optimizer.zero_grad()
  loss_dict = model(model_in, [target])
  losses = sum(loss for loss in loss_dict.values())
  p[image_number] = losses
  losses.backward()
  optimizer.step()
```

Figure 5: Data Augmentation during the network training

2.2 YOLO Object Detection

For using the YOLO-v3 model, we used yolo.h5 weights with the help of Object Detection from imageai.Detection library.[4]

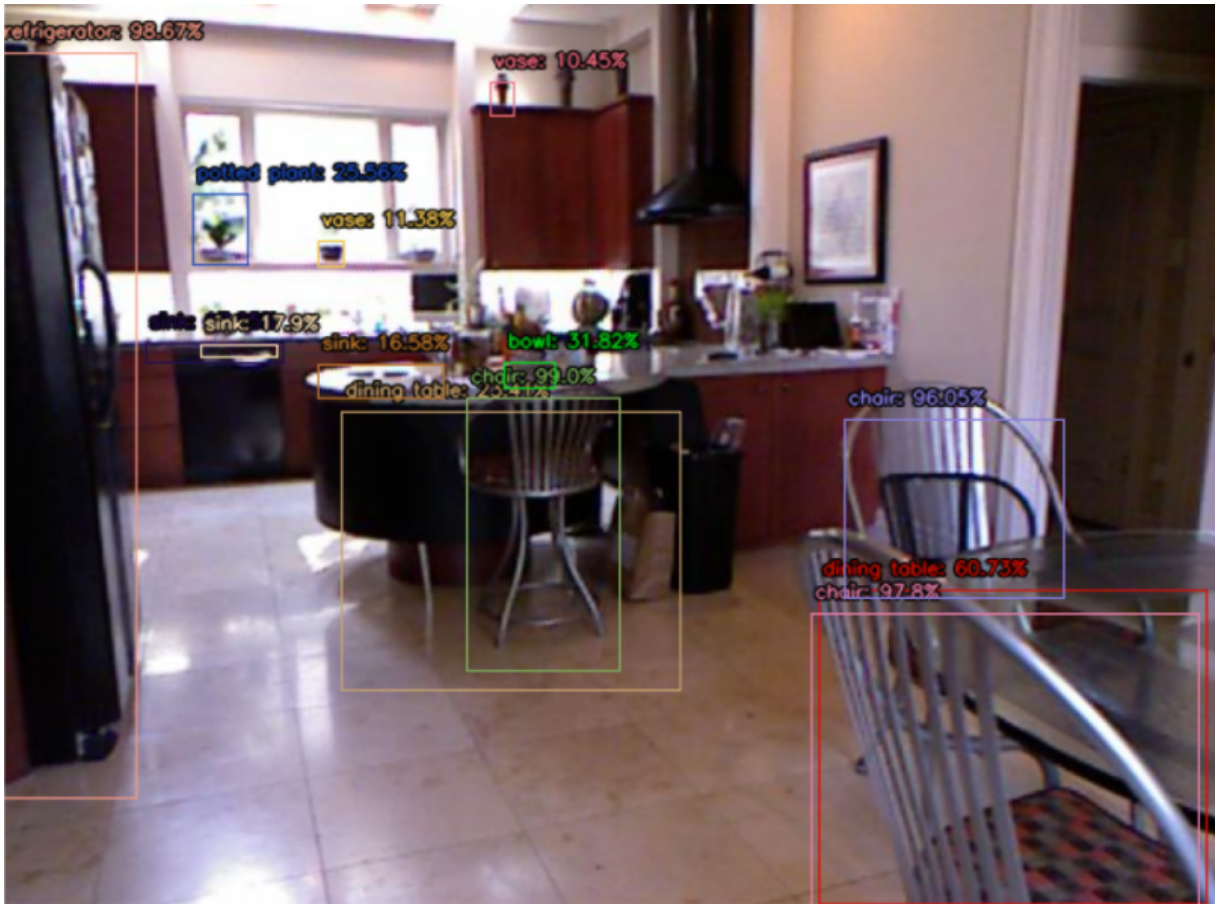

Figure 6: YOLO Object Detection on our data-set

3 Depth Estimation

3.1 Implementing U-Net from scratch

For implementing this part, we used Keras and designed the network based on its paper architecture.[1]

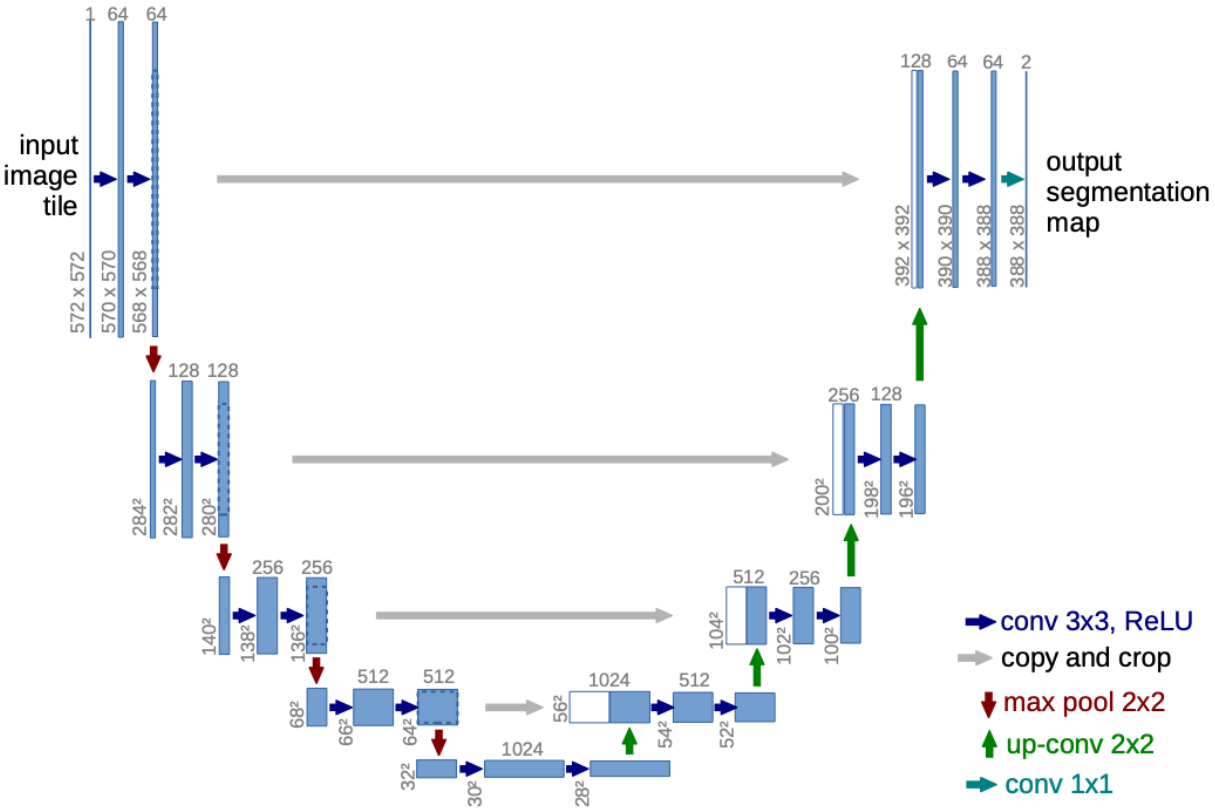

Figure 7: U-Net network

Then, we trained the designed network using each image as input and its depth matrix as output.

conv6 (Conv2D)	(None, 120, 160, 128	147584	conv5[0][0]
pool3 (MaxPooling2D)	(None, 60, 80, 128)	0	conv6[0][0]
conv7 (Conv2D)	(None, 60, 80, 256)	295168	pool3[0][0]
conv8 (Conv2D)	(None, 60, 80, 256)	590080	conv7[0][0]
pool4 (MaxPooling2D)	(None, 30, 40, 256)	0	conv8[0][0]
conv9 (Conv2D)	(None, 30, 40, 512)	1180160	pool4[0][0]
conv10 (Conv2D)	(None, 30, 40, 512)	2359808	conv9[0][0]
upconv1 (Conv2DTranspose)	(None, 60, 80, 512)	262656	conv10[0][0]
tf.concat (TFOpLambda)	(None, 60, 80, 768)	0	conv8[0][0] upconv1[0][0]
conv11 (Conv2D)	(None, 60, 80, 256)	1769728	tf.concat[0][0]
conv12 (Conv2D)	(None, 60, 80, 256)	590080	conv11[0][0]
upconv2 (Conv2DTranspose)	(None, 120, 160, 256	65792	conv12[0][0]
tf.concat_1 (TFOpLambda)	(None, 120, 160, 384	0	conv6[0][0] upconv2[0][0]
conv13 (Conv2D)	(None, 120, 160, 128	442496	tf.concat_1[0][0]
conv14 (Conv2D)	(None, 120, 160, 128	147584	conv13[0][0]
upconv3 (Conv2DTranspose)	(None, 240, 320, 128	16512	conv14[0][0]
tf.concat_2 (TFOpLambda)	(None, 240, 320, 160	0	conv4[0][0] upconv3[0][0]
conv15 (Conv2D)	(None, 240, 320, 64)	92224	tf.concat_2[0][0]
conv16 (Conv2D)	(None, 240, 320, 64)	36928	conv15[0][0]
upconv4 (Conv2DTranspose)	(None, 480, 640, 64)	4160	conv16[0][0]
tf.concat_3 (TFOpLambda)	(None, 480, 640, 96)	0	conv2[0][0] upconv4[0][0]
conv17 (Conv2D)	(None, 480, 640, 32)	27680	tf.concat_3[0][0]
conv18 (Conv2D)	(None, 480, 640, 32)	9248	conv17[0][0]
output (Conv2D)	(None, 480, 640, 1)	33	conv18[0][0]

Total params: 8,103,553
Trainable params: 8,103,553
Non-trainable params: 0

Figure 8: Our designed network layers

Finally, the result was not as smooth and accurate as we desired, so in the next part, we will combine the MiDaS network and our custom network to get better results for depth estimation.

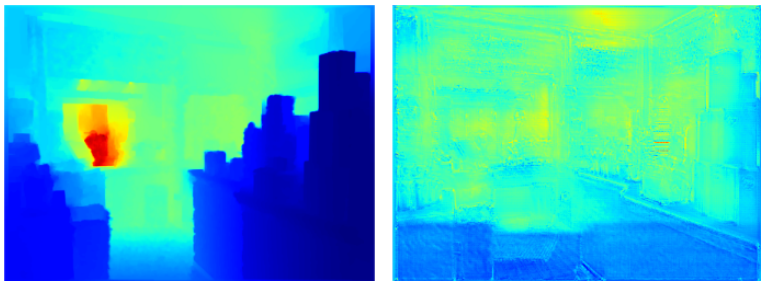

Figure 9: Our U-Net implemented network depth estimation result

3.2 Using Midas for Depth Estimation and our custom network for interpretation of its output

In this part, first, we load a large model with the highest accuracy. Then we design and train a network with two fully connected layers to transform the Midas output to our original depth map. [13] [12]

```
Model: "MiDaS-Depth"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input (InputLayer)          [(None, 4)]               0

 dense (Dense)               (None, 512)               2560

 dense_1 (Dense)             (None, 512)               262656

 dense_2 (Dense)             (None, 1)                 513

=================================================================
Total params: 265,729
Trainable params: 265,729
Non-trainable params: 0
_________________________________________________________________
```

Figure 10: Our own implemented MiDaS-Depth transformation network

For training this network, we tried different ways. First, we gave all pixels of a sample image in each step and then went to the next one. The problem with this method was that we saw jumps on network loss moving from image to image. So we decided to select 100 random pixels from each image of the data-set and then shuffle them all together. We also used mead, max, and min of the image pixels for our network. The result was good, as you can see below.

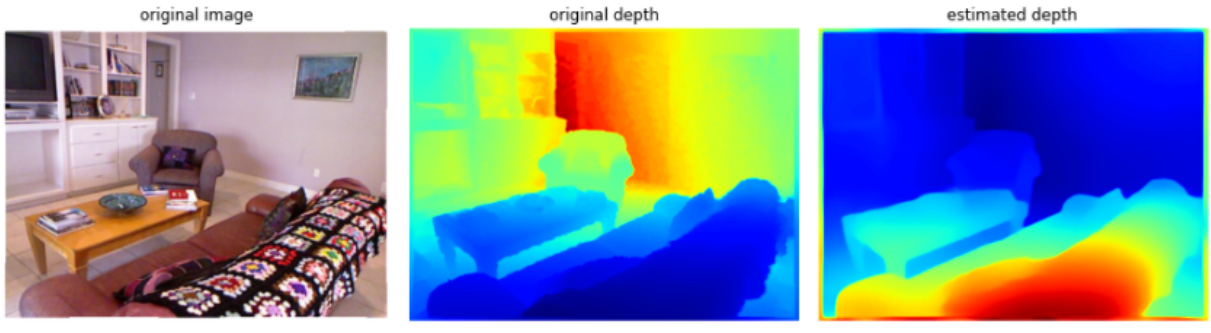

Figure 11: Result of depth estimation part

4 Error Estimation of joint network

4.1 Object Detection Network accuracy [7]

AP (Average precision) is a popular metric in measuring the accuracy of object detectors like Faster R-CNN, YOLO, etc. Average precision computes the average precision value for recall value over 0 to 1. But before explaining that, we will look into precision, recall, and IoU definition.

- Precision: measures how accurate is your predictions. i.e. the percentage of your predictions which are correct.
- Recall: measures how good you find all the positives.
- IoU (Intersection over union): measures the overlap between 2 boundaries. We use that to measure how much our predicted boundary overlaps with the ground truth (the real object boundary). In some data-sets, we predefine an IoU threshold (say 0.5) in classifying whether the prediction is a true positive or a false positive.

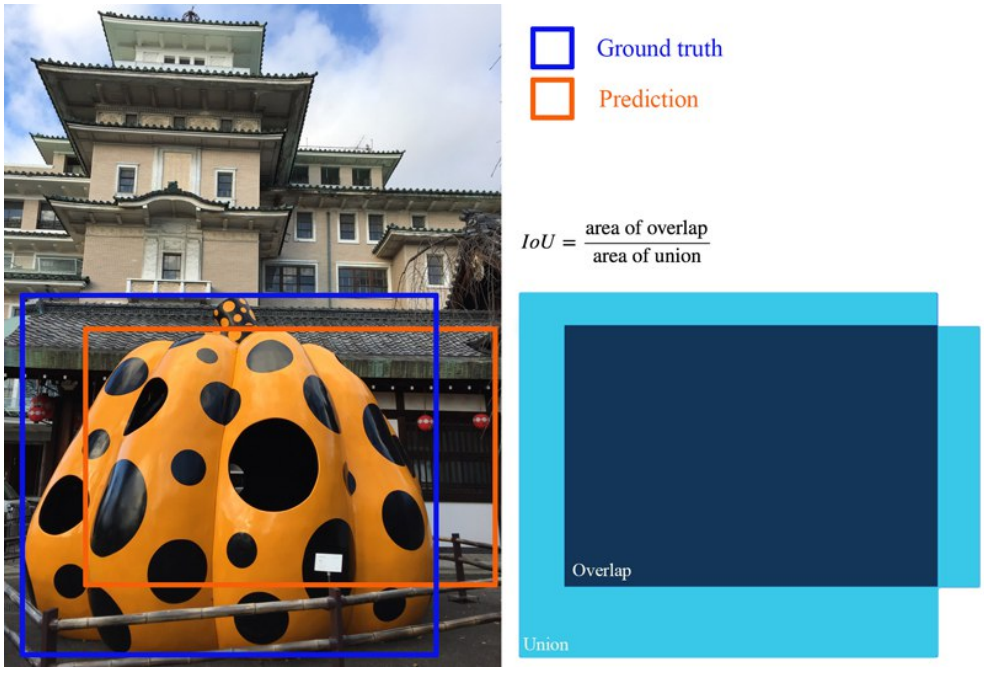

Figure 12: IoU

The general definition for the Average Precision (AP) is finding the area under the precision-recall curve.

$$AP = \int_0^1 p(r)dr$$

Precision and recall are always between 0 and 1. Therefore, AP falls within 0 and 1 also. Before calculating AP for the object detection, we often smooth out the zigzag pattern first.

4.1.1 Interpolated precision-recall curve

Graphically, at each recall level, we replace each precision value with the maximum precision value to the right of that recall level. The calculated AP value will be less suspectable to small variations in the ranking. Mathematically, we replace the precision value for recall $\hat{r}$ with the maximum precision for any $recall \geq \hat{r}$.

$$p_{interp}(r) = \max_{\tilde{r}>=r} p(\tilde{r})$$

4.1.2 Pascal VOC2008 AP

PASCAL VOC is a popular dataset for object detection. For the PASCAL VOC challenge, a prediction is positive if IoU ≥ 0.5. Also, if multiple detections of the same object are detected, it counts the first one as a positive while the rest as negatives.

In Pascal VOC2008, an average for the 11-point interpolated AP is calculated.

First, we divide the recall value from 0 to 1.0 into 11 points 0, 0.1, 0.2, ..., 0.9 and 1.0. Next, we compute the average of maximum precision value for these 11 recall values.

Here are the more precise mathematical definitions.

$$AP = \frac{1}{11} \sum_{r\in\{0.0,...,1.0\}} AP_r = \frac{1}{11} \sum_{r\in\{0.0,...,1.0\}} p_{interp}(r)$$

where

$$p_{interp}(r) = \max_{\tilde{r}>=r} p(\tilde{r})$$

When AP_r turns extremely small, we can assume the remaining terms to be zero. i.e. we don't necessarily make predictions until the recall reaches 100%. If the possible maximum precision levels drop to a negligible level, we can stop. For 20 different classes in PASCAL VOC, we compute an AP for every class and also provide an average for those 20 AP results.

However, this interpolated method is an approximation which suffers two issues. It is less precise. Second, it lost the capability in measuring the difference for methods with low AP. Therefore, a different AP calculation is adopted after 2008 for PASCAL VOC.

4.1.3 Pascal VOC2010-2012 AP

For later Pascal VOC competitions, VOC2010–2012 samples the curve at all unique recall values $(r_1, r_2, ...)$, whenever the maximum precision value drops. With this change, we are measuring the exact area under the precision-recall curve after the zigzags are removed.

No approximation or interpolation is needed. Instead of sampling 11 points, we sample $\mathrm{p}(r_i)$ whenever it drops and computes AP as the sum of the rectangular blocks.

$$AP = \sum (r_{n+1} - r_n) p_{interp}(r_{n+1})$$

$$p_{interp}(r_{n+1}) = \max_{\tilde{r} >= r_{n+1}} p(\tilde{r})$$

This definition is called the Area Under Curve (AUC).

4.1.4 COCO mAP

Latest research papers tend to give results for the COCO dataset only. In COCO mAP, a 101-point interpolated AP definition is used in the calculation. For COCO, AP is the average over multiple IoU (the minimum IoU to consider a positive match). AP@[.5:.95] corresponds to the average AP for IoU from 0.5 to 0.95 with a step size of 0.05. For the COCO competition, AP is the average over 10 IoU levels on 80 categories (AP@[.50:.05:.95]: start from 0.5 to 0.95 with a step size of 0.05). The following are some other metrics collected for the COCO data-set.

```
Average Precision (AP):
   AP                  % AP at IoU=.50:.05:.95 (primary challenge metric)
   AP^IoU=.50          % AP at IoU=.50 (PASCAL VOC metric)
   AP^IoU=.75          % AP at IoU=.75 (strict metric)
AP Across Scales:
   AP^small            % AP for small objects: area < 32^2
   AP^medium           % AP for medium objects: 32^2 < area < 96^2
   AP^large            % AP for large objects: area > 96^2
```

Figure 13: AP metrics definition

	backbone	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
Two-stage methods							
Faster R-CNN+++ [3]	ResNet-101-C4	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN [6]	ResNet-101-FPN	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI [4]	Inception-ResNet-v2 [19]	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM [18]	Inception-ResNet-v2-TDM	36.8	57.7	39.2	16.2	39.8	**52.1**
One-stage methods							
YOLOv2 [13]	DarkNet-19 [13]	21.6	44.0	19.2	5.0	22.4	35.5
SSD513 [9, 2]	ResNet-101-SSD	31.2	50.4	33.3	10.2	34.5	49.8
DSSD513 [2]	ResNet-101-DSSD	33.2	53.3	35.2	13.0	35.4	51.1
RetinaNet [7]	ResNet-101-FPN	39.1	59.1	42.3	21.8	42.7	50.2
RetinaNet [7]	ResNeXt-101-FPN	**40.8**	**61.1**	**44.1**	**24.1**	**44.2**	51.2
YOLOv3 608 × 608	Darknet-53	33.0	57.9	34.4	18.3	35.4	41.9

COCO for YOLOv3

Figure 14: YOLO accuracy [5]

4.2 Depth Estimation Network accuracy

Given a predicted depth image and the corresponding, ground truth, with $\hat{d}_p$ and d_p denoting the estimated and ground-truth depths respectively at pixel p, and T being the total number of pixels for which there exist both valid ground truth and predicted depth, the following metrics have been reported in literature: [10]

- Absolute Relative Error

$$\frac{1}{T}\sum_{p}\frac{|d_p - \hat{d}_p|}{d_p}$$

- Linear Root Mean Square Error (RMSE)

$$\sqrt{\frac{1}{T}\sum_{p}(d_p - \hat{d}_p)^2}$$

- Log scale invariant RMSE

$$\frac{1}{T}\sum_{p}(\log \hat{d}_p - \log d_p + \alpha(\hat{d}_p, d_p))^2$$

  where $\alpha(\hat{d}_p, d_p)$ addresses scale alignment

- Accuracy under a threshold

$$max(\frac{\hat{d}_p}{d_p}, \frac{d_p}{\hat{d}_p}) = \delta < th$$

  where th is a predefined threshold [11]

So we calculated all above metrics of our network on 100 images of the data-set. The results are below:

Figure 15: Accuracy metrics calculate on single image

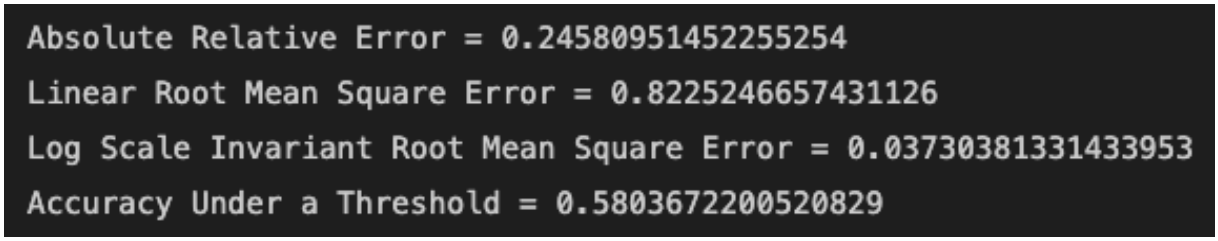

Figure 16: Accuracy metrics calculate for 100 images

Pay attention that the threshold on the above calculation was set on 1.25, that is a low number. If you increase the threshold, you will get higher accuracy.

4.3 Joint Network accuracy

To build up our joint network, we first get an image number from the user indicating which image should be tested from the data-set. Then, we will give the selected image to our depth estimation network.

After getting the depth estimation network result, by using the lower and upper bounds of allowed depth ($DepthBound_{up}, DepthBound_{down}$) taken from the user, we can put an appropriate mask on the image so that our object detection network can only detect objects in out desired bound.

For detecting objects of our data-set, we use the YOLO-v3 network, which has only 80 classes. So, this model detects everyday indoor objects like books, sinks, chairs, desks, beds, etc. We can not classify all kinds of objects inside the images. Therefore, for calculating our joint object detection network accuracy, we use the output of the object-detection network as ground truth in case of no depth limitation.

We have analyzed the references [3], and [6] introduced in the project. As our combined network is somehow different from their state-of-art, we have developed a new accuracy calculation method that will be more accurate for our use.

For doing so, we first calculate the $mean - depth$ of each object using the weighted average over pixels of its bounding box. Then, based on parameters $DepthBound_{up}$, $DepthBound_{down}$, and depth estimation network accuracy and using the idea of band-pass filters, we define an adjusting function.

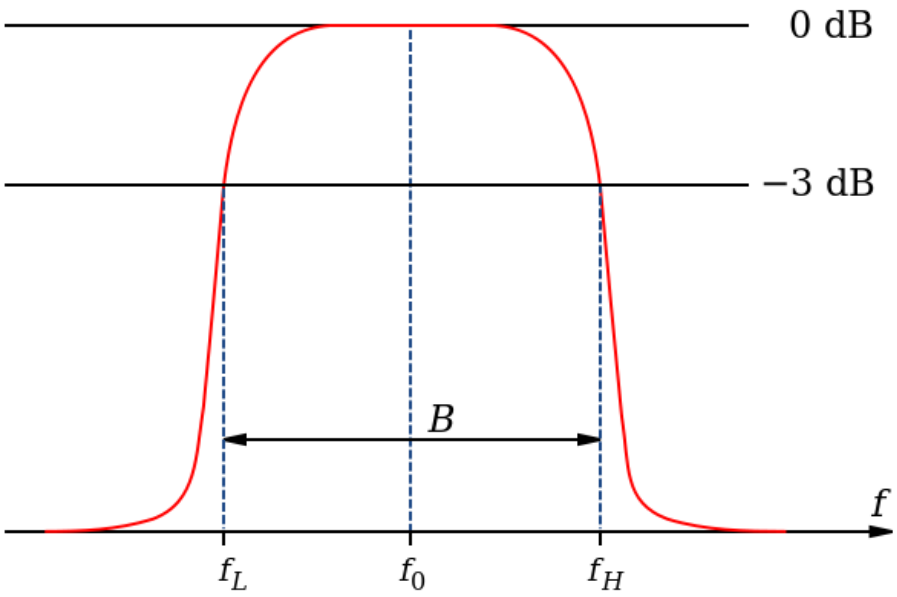

Figure 17: Band-pass filter

For our use, we put $DepthBound_{down}$ as f_L and $DepthBound_{up}$ as f_H. We also adjusted the Q factor of the band-pass filter according to depth estimation network accuracy.

In the end, for estimating the score of detected objects of the filtered image, we multiply the score of the corresponding object in explained ground-truth to the output of our adjusting function value for setting input as the $mean - depth$ of that image.

5 GUI (+)

We developed a GUI in google colab using Jupyter widgets as an extra part. In this part, we combined object-detection and depth-estimation networks from the last parts.

In our GUI, you can control three features:

- Image number you want to select from the data-set images
- Minimum prediction probability an object should have to be considered in output result
- Lower and upper bounds of the depth. Objects inside the range will be included

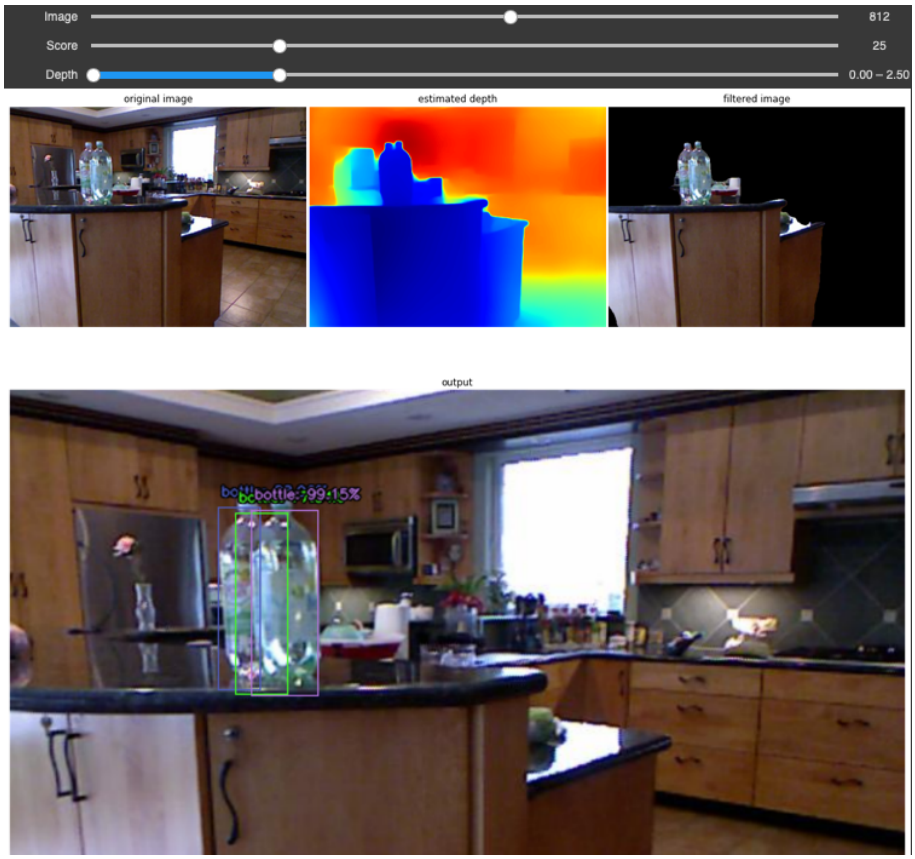

Figure 18: GUI (select image)

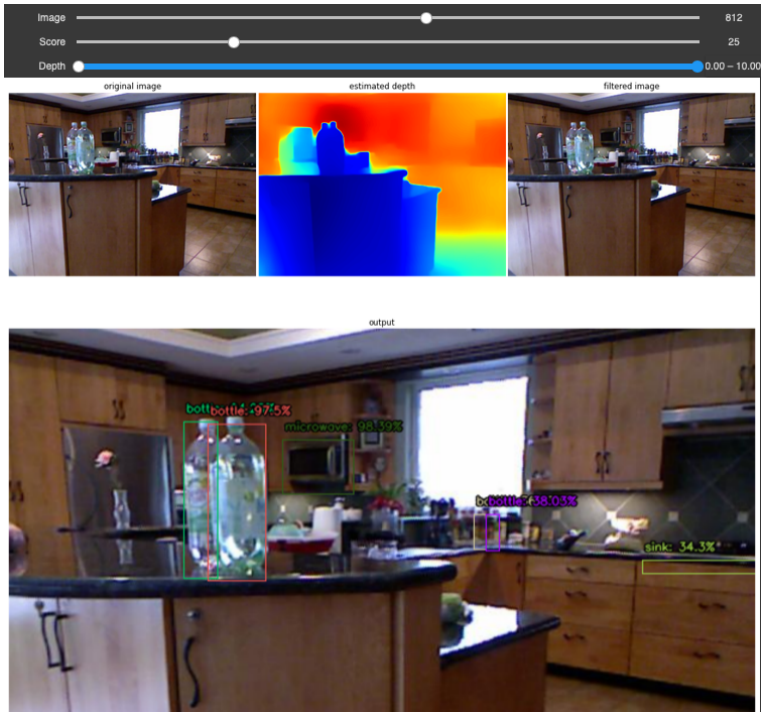

Figure 19: GUI (change depth upper bound)

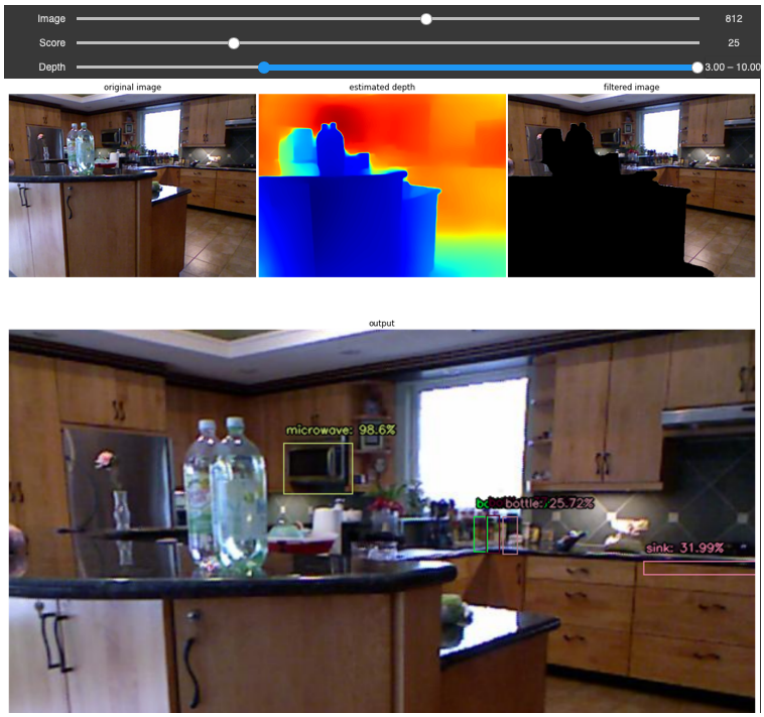

Figure 20: GUI (change depth lower bound)

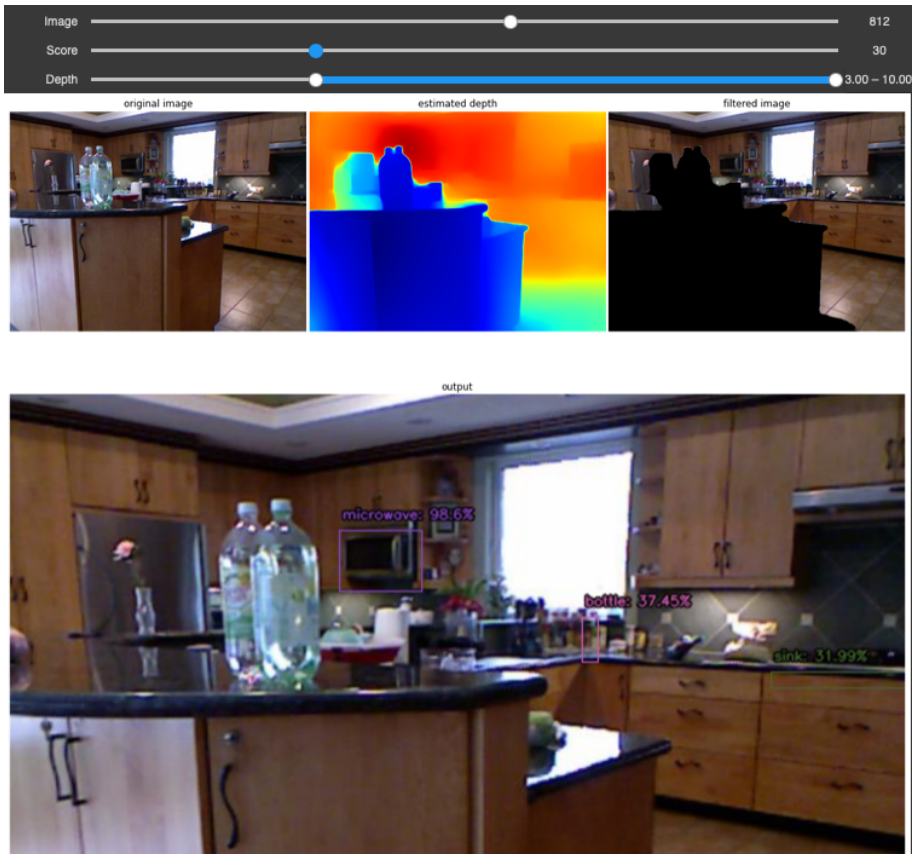

Figure 21: GUI (change score)

References

[1] U-Net.

[2] Faster RCNN Model.

[3] One Model To Learn Them All.

[4] YOLO-v3.

[5] Focal Loss for Dense Object Detection.

[6] BLEU.

[7] AP metric.

[8] Non Maximum Suppression theory and implementation in pytorch.

[9] Torchvision Object Detection Fine-tuning.

[10] Cesar Cadena, Yasir Latif, and Ian D Reid. Measuring the performance of single image depth estimation methods. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4150–4157. IEEE, 2016.

[11] David Eigen, Christian Puhrsch, and Rob Fergus. Depth map prediction from a single image using a multi-scale deep network. *Advances in neural information processing systems*, 27, 2014.

[12] René Ranftl, Alexey Bochkovskiy, and Vladlen Koltun. Vision transformers for dense prediction. *ArXiv preprint*, 2021.

[13] René Ranftl, Katrin Lasinger, David Hafner, Konrad Schindler, and Vladlen Koltun. Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2020.