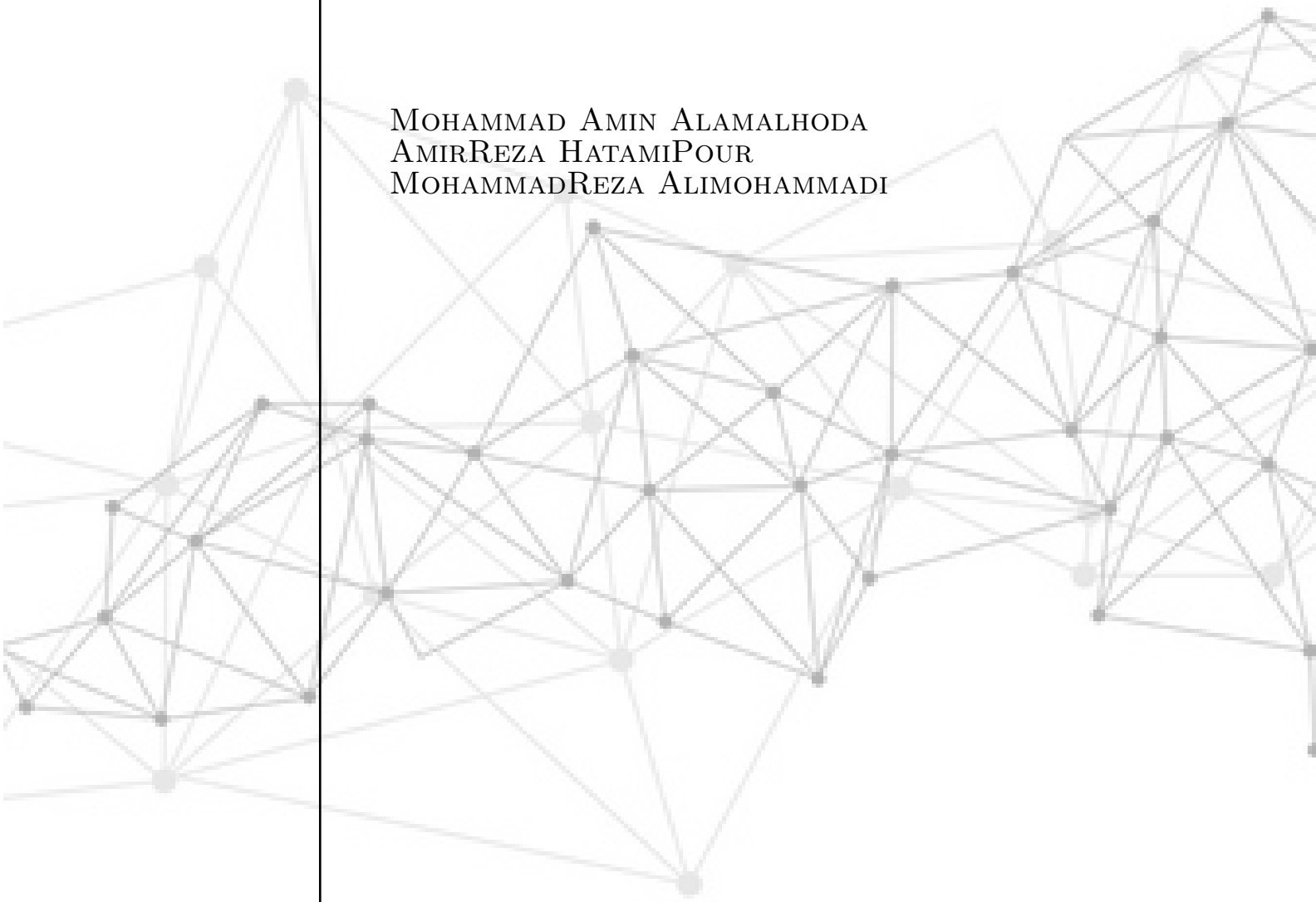
DL Course Project

Joint Depth Estimation and Object Detection

Mohammad Amin Alamalhoda
AmirReza HatamiPour
MohammadReza Alimohammadi



Sharif University of Technology $\boxed{\mathcal{SUT}}$

Contents

List of Figures

List of Tables

1 Git and Project Dependencies

1.1 Git

This Project is open source and is published on Github. You can watch it using this link.

You can use the following bash command for cloning this project:

```
$ git clone https://github.com/MohammadAminAlamalhoda/Deep-Object
```

If you don't have `git` installed on your device, you can use the following bash command:

- Linux

  ```
  $ sudo apt-get install git
  ```

- MacOS

  MacOS already have git installed, check its version using bash command below:

  ```
  $ git --version
  ```

  If you uninstalled it, you can install it using `brew`:

  ```
  $ brew install git
  ```

- Windows

  You can download source code of git and make-install it using this link.

1.2 Project Dependencies

This project needs the following stuff in order to be compiled successfully.

- PyTorch - Python Lib
- Torchvision - Python Lib
- OpenCV - Python Lib
- Matplotlib - Python Lib
- mat73 - Python Lib
- Os - Python Lib
- Sys - Python Lib
- Mask-RCNN- Torch Hub Model
- MiDaS - Torch Hub Model

You can install the required dependencies by running:

```
$ pip install -r requirements.txt
```

2 Project Structure

This project contains different files. We will explain them in the following.

2.1 train.ipynb

This jupyter notebook file contains the configs and essential properties for training the networks.

2.2 coco_eval.py and coco_utils.py

This files contain coco tools for image cropping and preparing images for object detection.

2.3 datas.py

This files contains dataloader classes and handles the loading and augmentation of the datas during training.

2.4 engine.py

This file contains the functions for training and evaluation of the networks.

2.5 models.py

All the model classes (Object Detector, Depth Estimator, and Concatener) are defined in this file.

2.6 transforms.py

Torchvision transfomation for augmenting the data set are defined in this file.

2.7 utils.py

This file contain some useful function such as model saver-loader, loggers, and ... .

2.8 GUI Directory

This directory contains files related to our application for loading and joint object detection and depth estimation of the datas.

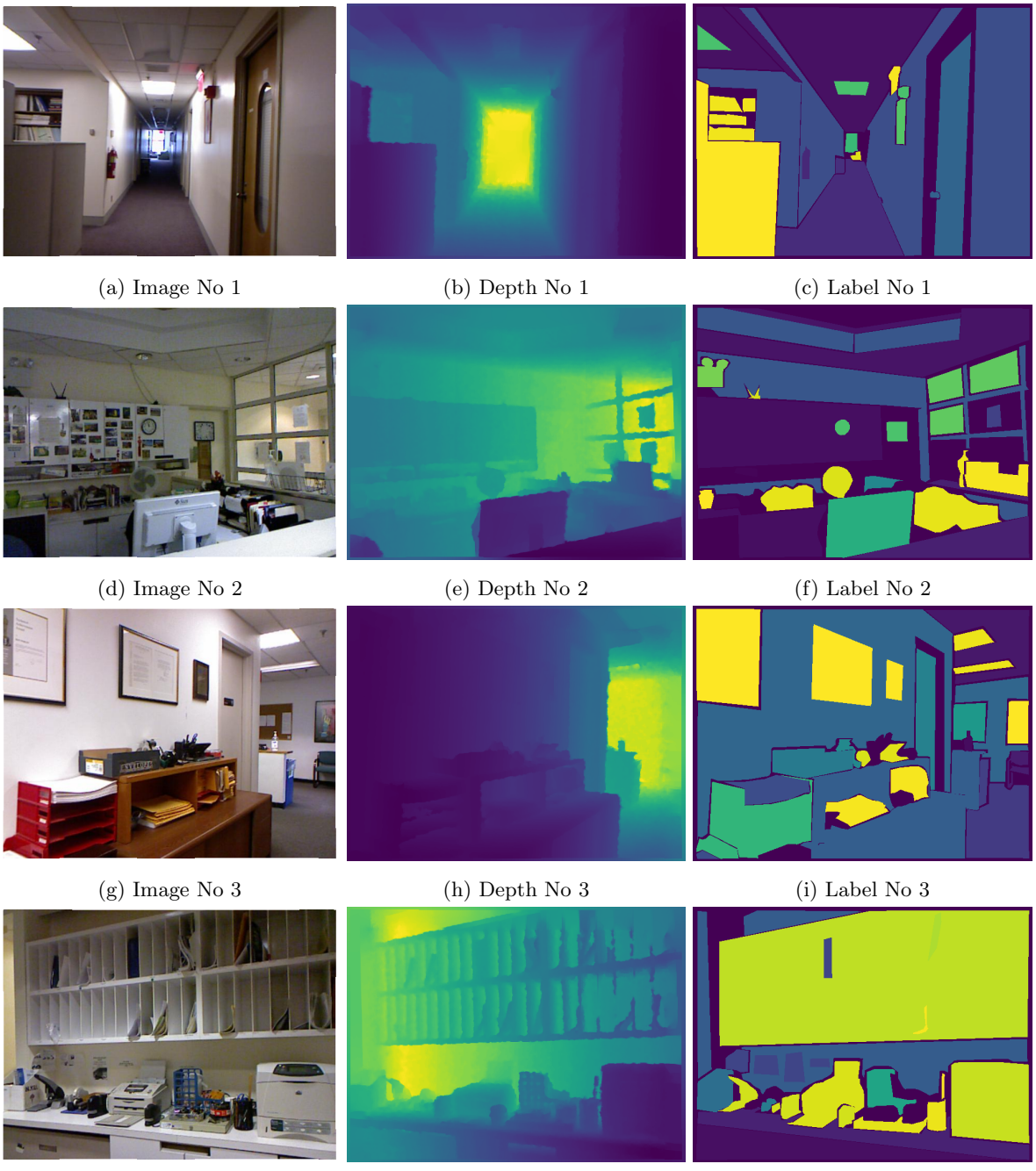
3 Datas

3.1 Converting Datas

We converted the datas which were in the .mat format to the .png for better RAM management. This way it is possible to load the images directly from the hard drive in each iteration.

3.2 Loading Datas

Figure 1 shows a sample from the dataset.



(a) Image No 1 (b) Depth No 1 (c) Label No 1

(d) Image No 2 (e) Depth No 2 (f) Label No 2

(g) Image No 3 (h) Depth No 3 (i) Label No 3

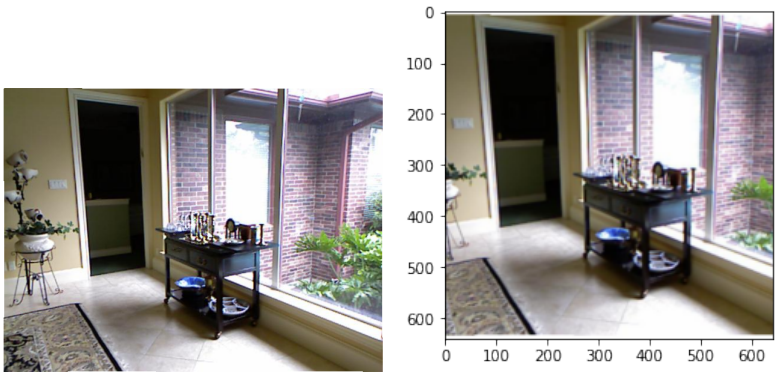
(j) Image No 4 (k) Depth No 4 (l) Label No 4

Figure 1: Some Sample Images from The Dataset

As can be seen in Figure 1, dataset contains image, label, and depth.

3.3 Data Augmentation

We augmented the datas by resizing to 640, random cropping an 640×640 square, and random horizontal flip. Some of the Augmented Images are plotted in Figure 2. It is noteworthy to mention that all the datas(images, labels, and depths) which have belong to a specific scene, should be augmented by same properties.



(a) Image (b) Augmented Image

Figure 2: An Image and its Augmented Version

Transforms are done using *torchvisoin.datas.trasnforms*.

3.4 Data Loader

We used *torch data loaders* for managing datas and augmentation. You can find this files in datas.py file.

3.5 Loading .mat Dataset

Due to RAM shortage, we didn't load the datas on the RAM. All the images, labels and depths in .mat file were converted to .png images and the loaded directly from the hard drive during training. This was done using function `mat2png` which can be find in `utils.py`.

4 Models

4.1 Object Detector

We used pretrained *Mask-RCNN* as the object detector network and fine tuned it using our dataset. You can find the paper of *Mask-RCNN* using this link.

4.1.1 Architecture

This network develops, for instance segmentation. Instance segmentation is challenging because it requires the correct detection of all objects in an image while precisely segmenting each instance. It therefore combines elements from the classical computer vision tasks of object detection, where the goal is to classify individual objects and localize each using a bounding box, and semantic segmentation, where the goal is to classify each pixel into a fixed set of categories without differentiating object instances. Mask R-CNN, extends Faster R-CNN by adding a branch for predicting segmentation masks on each Region of Interest (RoI), in parallel with the existing branch for classification and bounding box regression. You can see an example of box regression in the Figure 3.

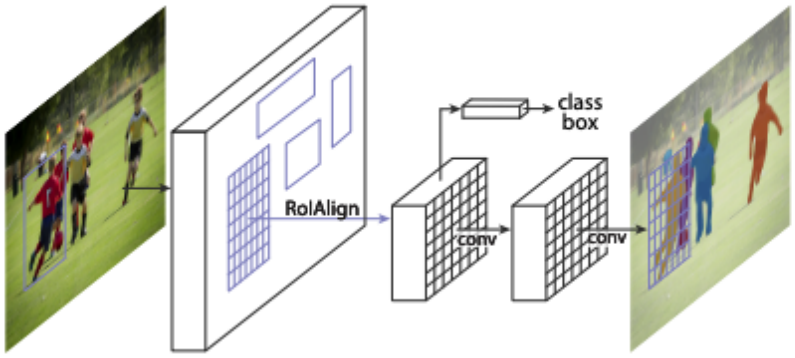


Figure 3: Bounding Box Regression in the Mask-RCNN

The mask branch is a small FCN applied to each RoI, predicting a segmentation mask in a pixel-to-pixel manner. Mask R-CNN is simple to implement and train given the Faster R-CNN framework, which facilitates a wide range of flexible architecture designs. Additionally, the mask branch only adds a small computational overhead, enabling a fast system and rapid experimentation. Faster RCNN was not designed for pixel-to-pixel alignment between network inputs and output. To fix the misalignment, proposed a simple, quantization-free layer, called RoIAlign, that faithfully preserves exact spatial locations. Despite being a seemingly minor change, RoIAlign has a large impact: it improves mask accuracy by relative 10% to 50%, showing bigger gains under stricter localization metrics. Second, we found it essential to decouple mask and class prediction: we predict a binary mask for each class independently, without competition among classes, and rely on the network's RoI classification branch to predict the category.

4.1.2 Concept

Mask R-CNN is conceptually simple: Faster R-CNN has two outputs for each candidate object, a class label, and a bounding-box offset; to this, we add a third branch that outputs the object mask. Mask R-CNN is thus a natural and intuitive idea. But the additional mask output is distinct from the class and box outputs, requiring extraction of an object's much finer spatial layout. Next, we introduce the key elements of Mask R-CNN, including pixel-to-pixel alignment, which is the main missing piece of Fast/Faster R-CNN.

4.1.3 Computation Procedure

Mask R-CNN adopts the same two-stage procedure, with an identical first stage (RPN). In parallel to predicting the class and box offset in the second stage, Mask R-CNN also outputs a binary mask for each RoI. This contrasts with most recent systems, where classification depends on mask predictions. Our approach follows the spirit of Fast R-CNN that applies bounding-box classification and regression in parallel (which turned out to largely simplify the multi-stage pipeline of original R-CNN). Formally, during training, we define a multi-task loss on each sampled RoI as:

$$L \quad = L_{cs} + L_{box} + L_{mask}$$

The classification loss L_{cs} and bounding-box L_{bo}. The mask branch has a Km^2-dimensional output for each RoI, which encodes K binary masks of resolution m × m, one for each of the K classes. To this, apply a per-pixel sigmoid and define L_{mask} as the average binary cross-entropy loss, for an RoI associated with ground-truth class k, L_{mask} is only defined on the k^{th} mask (other mask outputs do not contribute to the loss).

L_{mask} allows the network to generate masks for every class without competition among classes; They relied on the dedicated classification branch to predict the class label used to select the output mask. This decouples mask and class prediction. This is different from common practice when applying FCNs to semantic segmentation, which typically uses a per-pixel softmax and a multinomial cross-entropy loss. In that case, masks across classes compete; they do not have a per-pixel sigmoid and a binary loss.

You can see an overview of *Mask-RCNN* in the Figure 4

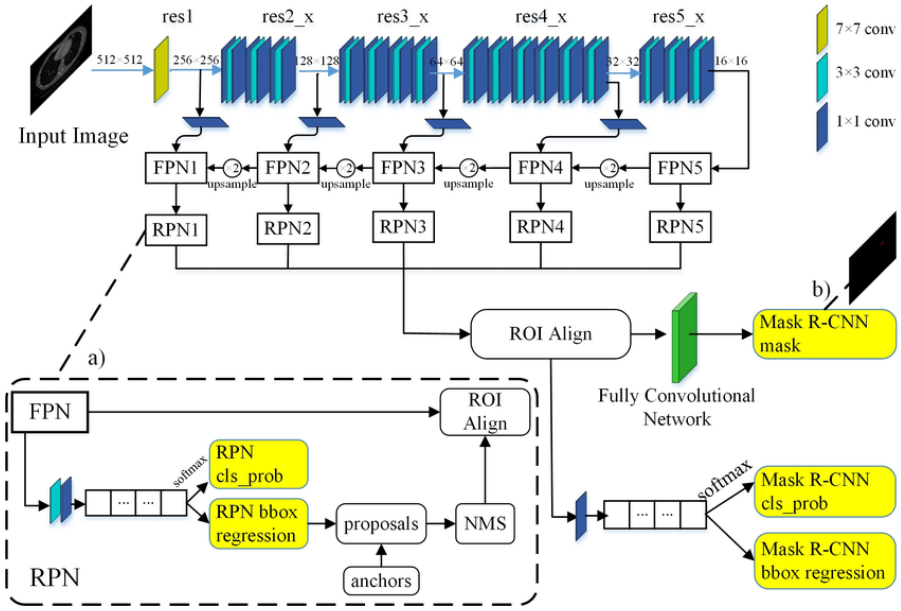


Figure 4: A Schematic of the Mask-RCNN

4.1.4 Implementation Details

- **Training**: As in Fast R-CNN, an RoI is considered positive if it has IoU with a ground-truth box of at least 0.5 and negative otherwise. The mask loss L_{mask} is defined only on positive RoIs. The mask target is the intersection between an RoI and its associated ground-truth mask. They adopt image-centric training. Images are resized such that their scale (shorter edge) is 800 pixels. Each mini-batch has 2 images per GPU and each image has N sampled RoIs, with a ratio of 1:3 of positive to negatives. N is 64 for the C4 backbone and 512 for FPN. We train on 8 GPUs (so effective mini batch size is 16) for 160k iterations, with a learning rate of 0.02 which is decreased by 10 at the 120k iteration. They use a weight decay of

0.0001 and momentum of 0.9. With ResNeXt, they train with 1 image per GPU and the same number of iterations, with a starting learning rate of 0.01. The RPN anchors span 5 scales and 3 aspect ratios. For convenient ablation, RPN is trained separately and does not share features with Mask R-CNN, unless specified. For every entry in this paper, RPN and Mask R-CNN have the same backbones and so they are shareable.

- **Inference**: At test time, the proposal number is 300 for the C4 backbone and 1000 for FPN. We run the box prediction branch on these proposals, followed by non-maximum suppression. The mask branch is then applied to the highest scoring 100 detection boxes. Although this differs from the parallel computation used in training, it speeds up inference and improves accuracy (due to the use of fewer, more accurate RoIs). The mask branch can predict K masks per RoI, but we only use the k^{th} mask, where k is the predicted class by the classification branch. The m×m floating-number mask output is then resized to the RoI size, and binarized at a threshold of 0.5. Note that since they only compute masks on the top 100 detection boxes, Mask R-CNN adds a small overhead to its Faster R-CNN counterpart (eg. 20% on typical models).
- **Experiments**: Instance Segmentation They perform a thorough comparison of Mask R-CNN to the state of the art along with comprehensive ablations on the COCO dataset. they report the standard COCO metrics including AP (averaged over IoU thresholds), AP50, AP75, and APS, APM, APL (AP at different scales). Unless noted, AP is evaluating using mask IoU.

You can see some of the outputs of the *Mask-RCNN* in Figure 5 and Instance segmentation mask APs in Table 1.

Figure 5: Some Results of Mask R-CNN on COCO Test Images, using ResNet-101-FPN and Running at 5 fps, with 35.7 Mask AP

Table 1: Instance Segmentation Mask AP on COCO Test-Dev

	Back-Bone	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
MNC	ResNet-101-C4	24.6	44.3	24.8	4.7	25.9	43.6
FCIS+OHEM	ResNet-101-C5-dilated	29.2	49.5	-	7.1	25.9	43.6
FCIS+++ + OHEM	ResNet-101-C5-dilated	33.6	53.5	-	-	-	-
Mask R-CNN	ResNet-101-C4	33.1	54.9	34.8	12.1	25.6	51.1
Mask R-CNN	ResNet-101-FPN	35.7	58.0	37.8	15.5	38.1	52.4
Mask R-CNN	ResNetXt-101-FPN	37.1	60.0	39.4	16.9	39.9	53.5

4.1.5 Fine-Tuning

As mentioned earlier, we fine tuned the model by using our datas. You can see the training and validation loss plots in Figures 6 and 7.

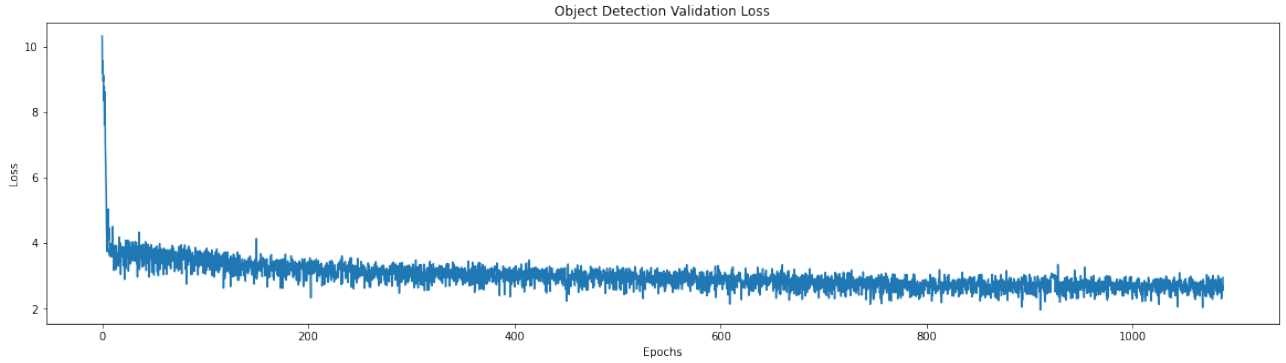


Figure 6: Training Loss of the Object Detector Model

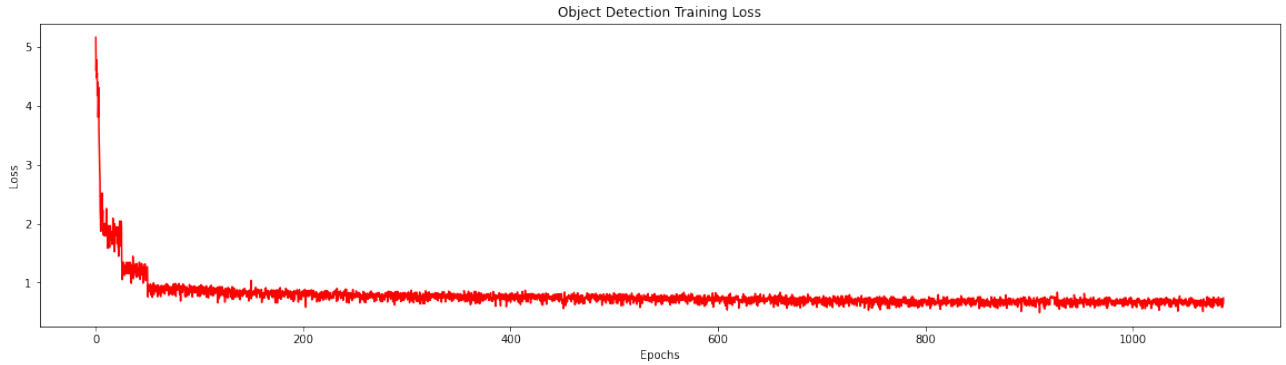


Figure 7: Validation Loss of the Object Detector Model

4.1.6 Evaluation

Evaluating object detection models is not straightforward because each image can have many objects and each object can belong to different classes. This means that we need to measure if the model found all the objects and also a way to verify if the found objects belong to the correct class. This means that an object detection model needs to accomplish two things

- Find all the objects in an image
- Check if the found objects belong to the correct class

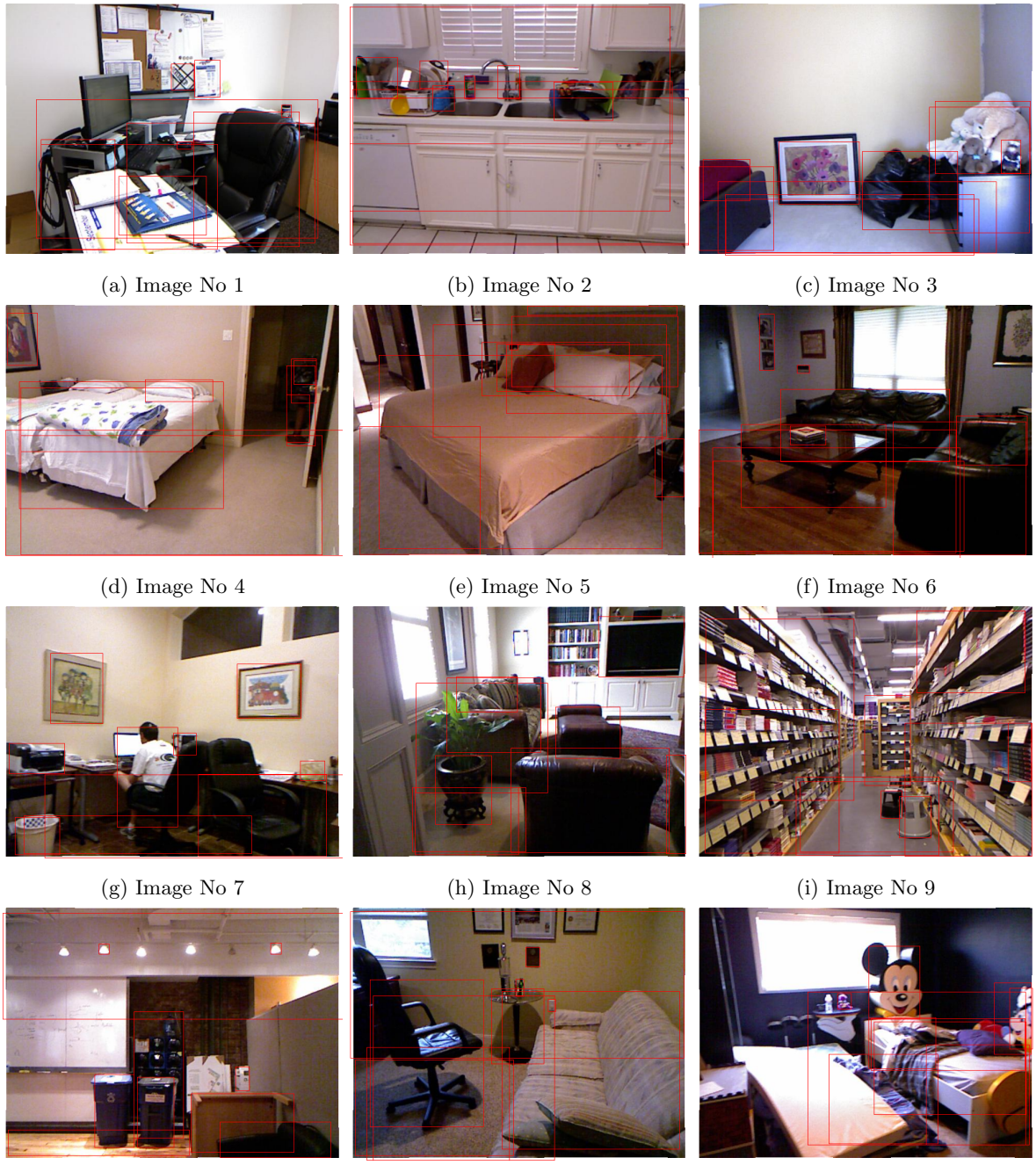
Therefore, we used different metrics for evaluating the object detector model which are written in Table 2:

Table 2: Object Detection Metrics

Metric	Initial Value-Training	Final Value-Training	Initial Value-Validation	Final Value-Validation
Total loss	8.892	2.7060	4.323	1.413
Classifier Loss	3.902	1.4148	1.032	0.242
Box Reg Loss	2.023	0.6253	1.231	0.321
Mask Loss	1.834	0.4961	0.891	0.214
Objectness Loss	0.673	0.0435	0.413	0.001

4.1.7 Test Images

You can see some test images in the Figure 8.



(a) Image No 1 (b) Image No 2 (c) Image No 3

(d) Image No 4 (e) Image No 5 (f) Image No 6

(g) Image No 7 (h) Image No 8 (i) Image No 9

(j) Image No 10 (k) Image No 11 (l) Image No 12

Figure 8: Output of Object Detector Network for Some Test Images

4.2 Depth Estimator

We used pretrained *Intelisl MiDaS* as the depth estimation network and fine tuned it using our dataset. You can find the paper of *MiDaS* using this link.

Depth estimation of an image predicts the order of objects (if the image was expanded in a 3D format) from the 2D image itself. It is an unequivocally difficult task since getting annotated data and datasets specializing in this area was a mammoth task itself. MiDaS is a machine learning model that estimates depth from an arbitrary input image. More specifically, MiDaS computes relative inverse depth from a single image.

4.2.1 Architecture

Various datasets containing depth information are not compatible in terms of scale and bias. This is due to the diversity of measuring tools, including stereo cameras, laser scanners, and light sensors. MiDaS introduces a new loss function that absorbs these diversities, thereby eliminating compatibility issues and allowing multiple data sets to be used for training simultaneously. MiDaS developers firmly assert that training models on a single dataset will not be robust when dealing with problem statements encompassing real-life issues. When models used in real-time are created, they should be robust enough to deal with as many situations and outliers as possible. Therefore, they decide to train their model on multiple datasets, as shown in the table below. Therefore, it can estimate the depth of images in various conditions and environments. The idea seems very ingenious, but they had to carefully devise loss functions and consider the challenges accompanying the choice of using multiple datasets. Since these datasets had different representations of depth estimation to varying degrees, an inherent scale ambiguity, and shift ambiguity come up, as the paper's authors addressed. Now, since the datasets in all probability would follow distributions different from each other. Hence the issues are pretty much expected. However, the authors propose solutions to each of the challenges. The end product was a robust depth estimator, which was as efficient as accurate. Figure 9 illustrates some results shown in the paper. The idea of cross dataset learning isn't new, but the complications that arise from getting the ground truths to a common output space are extremely tough to get by. However, the paper explains each step extensively, starting with the intuition right to the mathematical definition of the losses used.

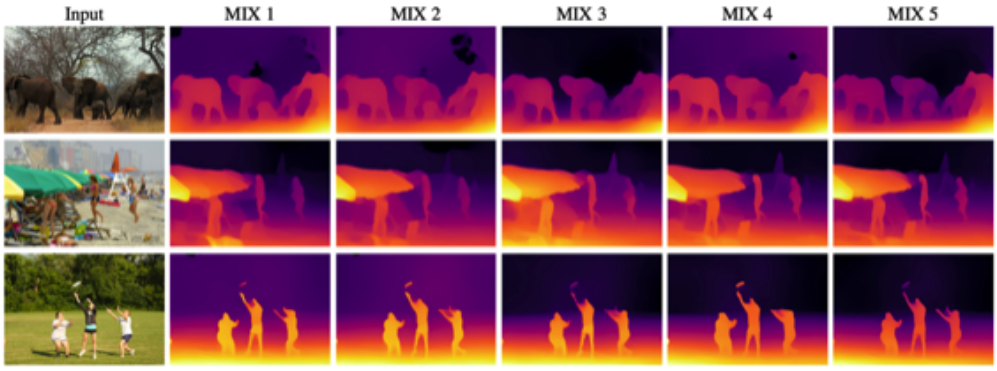


Figure 9: Comparison of Depth Estimation Models Trained on Different Combinations of Datasets

4.2.2 Loss Function

MiDaS has developed a novel loss functions that are invariant to the major sources of incompatibility between datasets, including unknown and inconsistent scale and baselines. MiDaS losses enable training on data that was acquired with diverse sensing modalities such as stereo cameras with potentially unknown calibration, laser scanners, and structured light sensors. MiDaS also quantify the value of a variety of existing datasets for monocular depth estimation and explore optimal strategies for mixing datasets during training. They show that a principled approach based on multi-objective optimization leads to improved results compared to a naive mixing strategy. As mentioned above, training models for monocular depth estimation on diverse datasets presents a challenge because the ground truth comes in different forms, see Figure 10 and Equations 1, 2, 3, and 4. So, MiDaS designers had to design a loss function that is flexible enough to handle diverse sources of data while making optimal use of all available information. More specifically they proposed loss functions must solve theses major challenges:

- Inherently different representations of depth: direct vs. inverse depth representations.
- Scale ambiguity: for some data sources, depth is only given up to an unknown scale.
- Shift ambiguity: some datasets provide disparity only up to an unknown scale and global disparity shift that is a function of the unknown baseline and a horizontal shift of the principal points due to post-processing

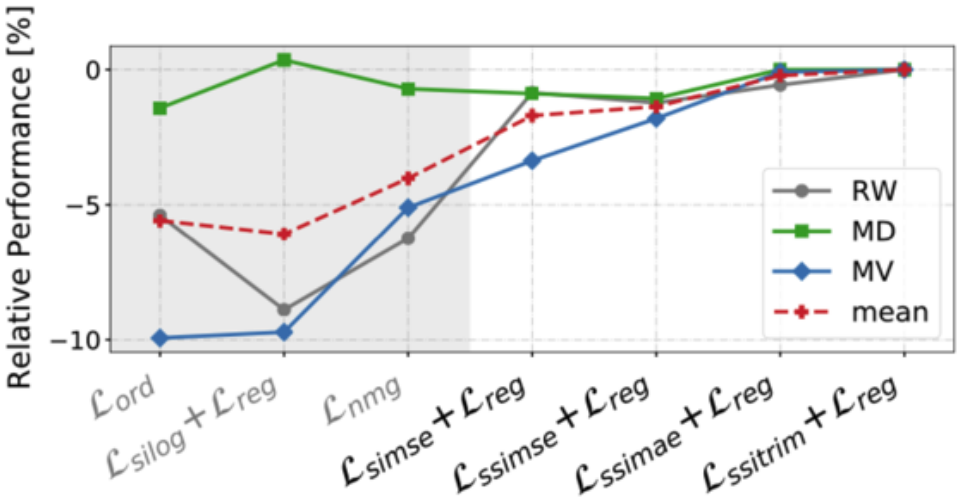


Figure 10: Different Midas Losses and Their Performance on Different Datasets

$$\mathcal{L}_{ssi} = \frac{1}{2M}\sum_{i=1}^{M} p(\hat{\mathbf{d}} - \hat{\mathbf{d}}^*) \tag{1}$$

$$\mathcal{L}_{reg} = \frac{1}{M}\sum_{k=K}^{M}\sum_{i=1}^{M}(|\nabla_x R_i^k| + |\nabla_y R_i^k|) \tag{2}$$

$$\mathcal{L}_{silog} = min\ \frac{1}{2M}\sum_{i=1}^{M}(\log(e^2 z_i) - \log z_i^*)^2 \tag{3}$$

$$\mathcal{L}_{ssitrim} = \frac{1}{2M}\sum_{j=1}^{U_m} p_{mae}(\hat{\mathbf{d}}_j - \hat{\mathbf{d}}_j^*) \tag{4}$$

$$\mathcal{L}_l = \frac{1}{N_l}\sum_{n=1}^{N_l} \mathcal{L}_{ssi}\left(\hat{\mathbf{d}}^n, \left(\hat{\mathbf{d}}^*\right)^n\right) + \alpha\mathcal{L}_{reg}\left(\hat{\mathbf{d}}^n, \left(\hat{\mathbf{d}}^*\right)^n\right) \tag{5}$$

Due to the performance of $\mathcal{L}_{ssi}$ and $\mathcal{L}_{reg}$ (Figure 10), MiDaS designers used $\mathcal{L}_l$ as the loss term (Equation 5).

4.2.3 Encoders

MiDaS designers have tested ResNet-50, ResNet-101, DenseNet-161, ResNeXt-101, and ResNeXt-101-WSL. All these encoders were pretrained on ImageNet. They observe that a significant performance boost is achieved by using better encoders. Higher-capacity encoders perform better than the baseline. In general, we find that ImageNet performance of an encoder is a strong predictor for its performance in monocular depth estimation. This is very interesting, since advancements made in image classification can directly yield gains in robust monocular depth estimation. You can see relevant performance of different encoders in Figure 11

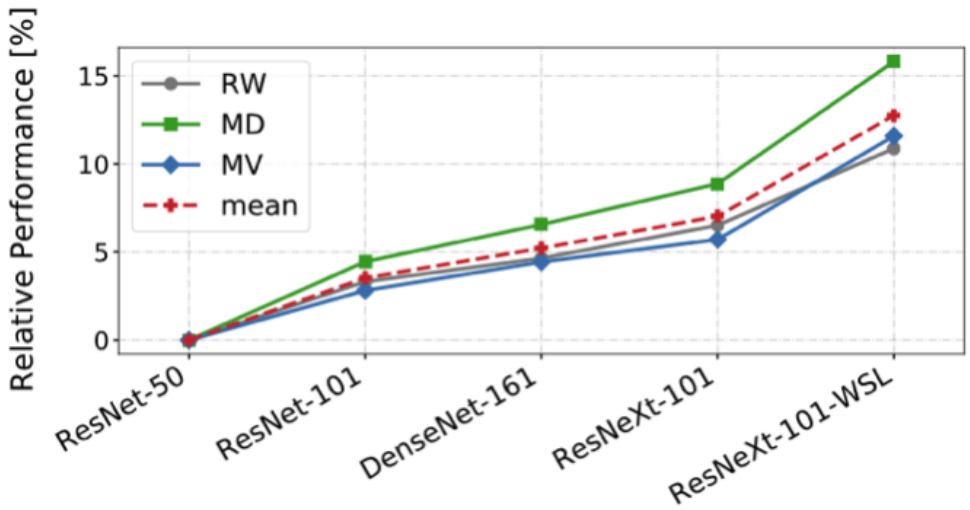


Figure 11: Relative Performance of Different Encoders Across Datasets (higher is better). ImageNet Performance of an Encoder is Predictive of its Performance in Monocular Depth Estimation.

4.2.4 Fine-Tuning

As mentioned earlier, we fine tuned the model by using our datas. You can see the training loss plot in Figure 12.

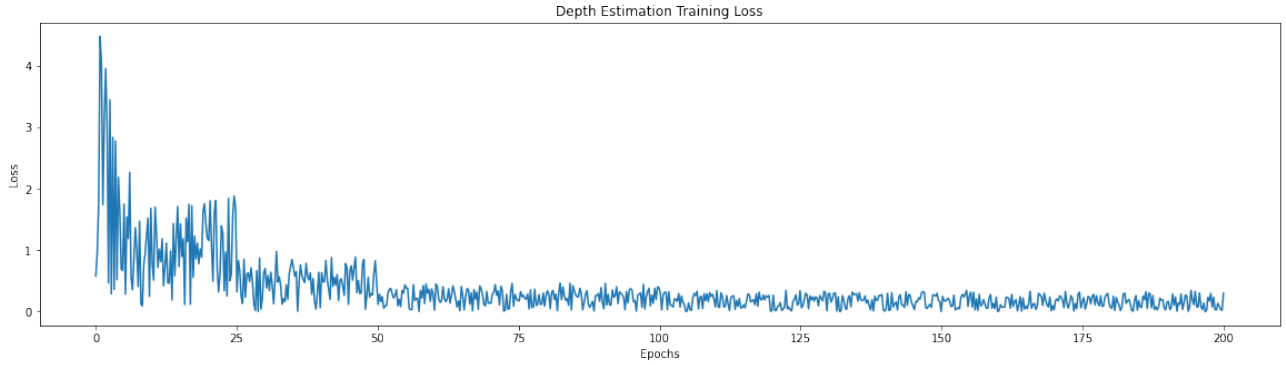


Figure 12: Training Loss of the Depth Estimator Model

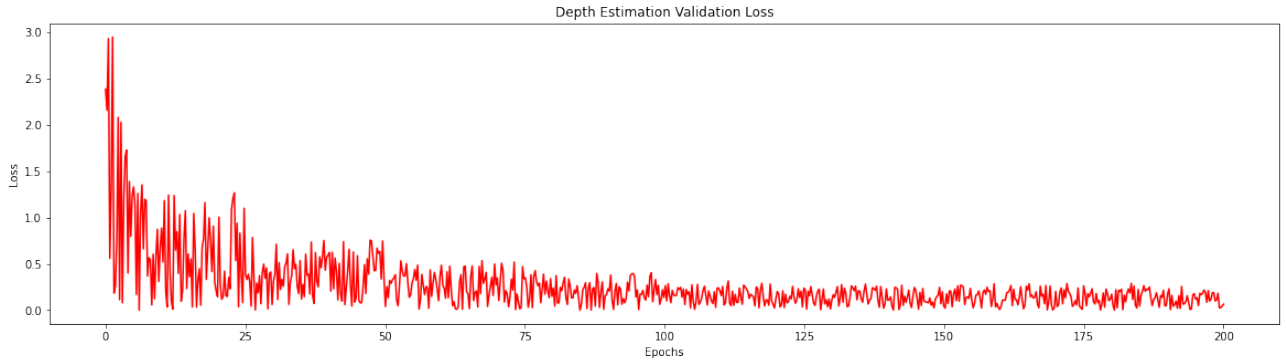


Figure 13: Vlidatoin Loss of the Depth Estimator Model

4.2.5 Evaluation

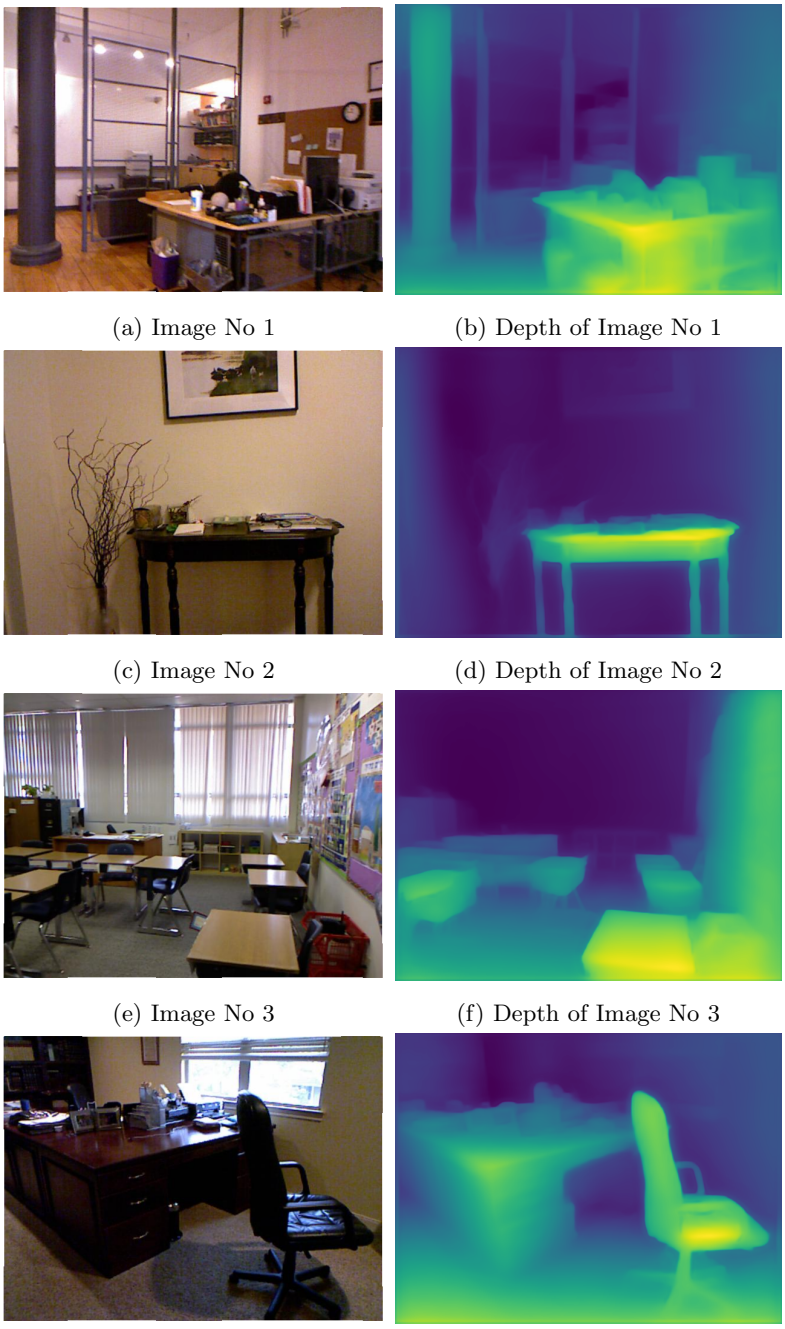
Estimating depth of different pixels in an image is a regression problem, so we used MSE(mean squad error) as loss. You can see this loss values for training and validation in the Table 3.

Table 3: Depth Estimator Metrics

Metric	Initial Value-Training	Final Value-Training	Initial Value-Validation	Final Value-Validation
MSE loss	4.592	0.2060	6.234	0.124

4.2.6 Test Images

You can see some test images in the Figure 14.



(a) Image No 1 (b) Depth of Image No 1

(c) Image No 2 (d) Depth of Image No 2

(e) Image No 3 (f) Depth of Image No 3

(g) Image No 4 (h) Depth of Image No 4

Figure 14: Output of Depth Estimator Network for Some Test Images

5 Concatenator

We used two different methods for concatenating the object detector and depth estimator:

5.1 First Method

In the first method, we used an auto-encoder network which was placed after the depth estimator and before the object detector. This AE's input was the estimated depth and was the detected objects and masks. This network was learned when both the other networks were frozen. You can see a schematic of this AE in Figure 15.

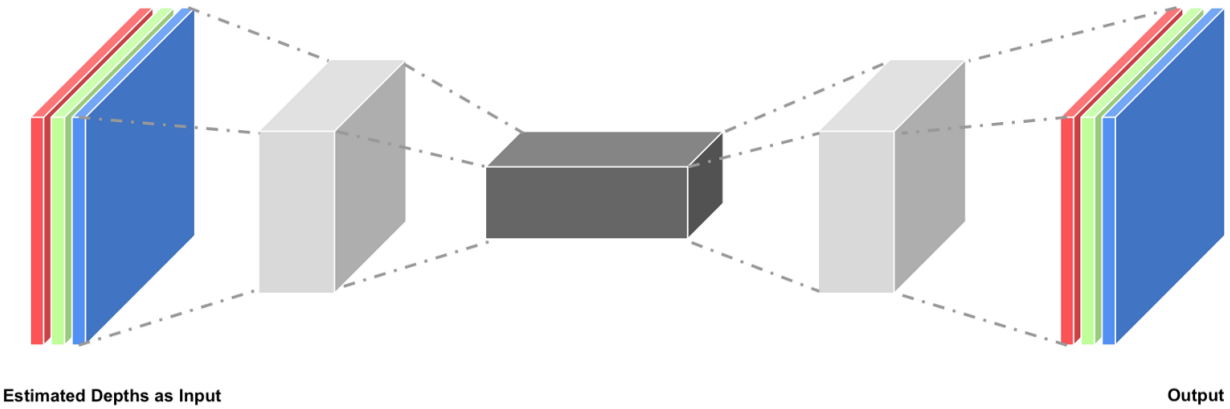


Figure 15: Schematic of the AE used for Concatenating

This method can perform well if we have large dataset and very fast computing hardware, but because of the small dataset and slow computing hardware, we couldn't fully train this model.

5.1.1 Evaluation

We used BLEU and perplexity metrics for evaluating functionality of the concatenator. You can see the perplexity of the training and validation in the Figure 16 and definition of the perplexity of a probability distribution in 6.

$$PP(p) \quad = 2^{H(p)} = 2^{-\sum p(x)} \tag{6}$$

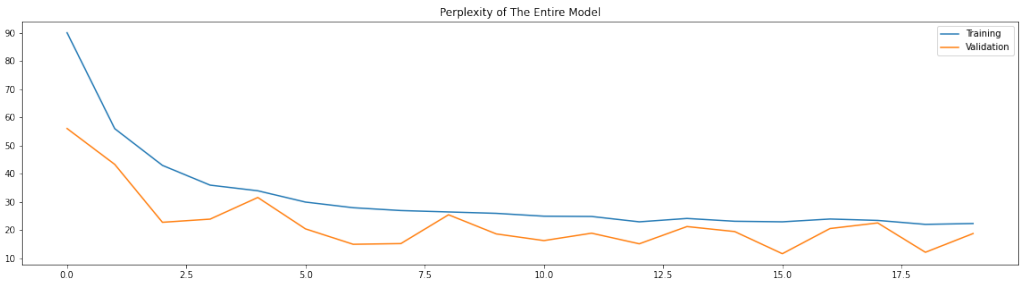
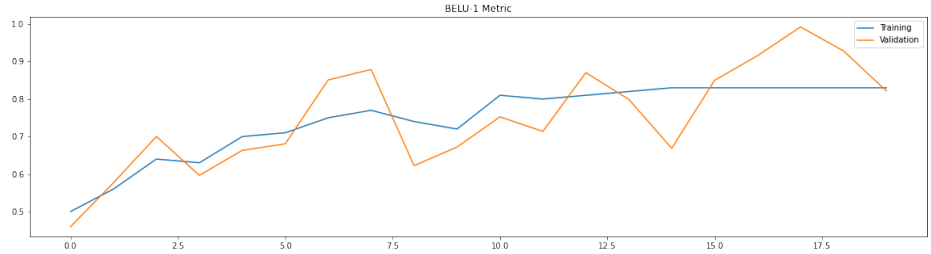
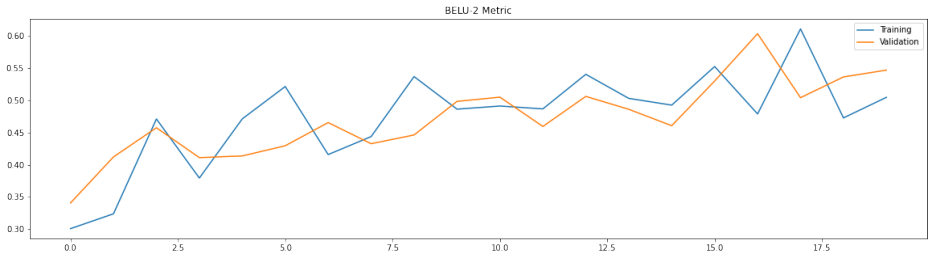


Figure 16: Perplexity of Training and Validation (Lower is better)

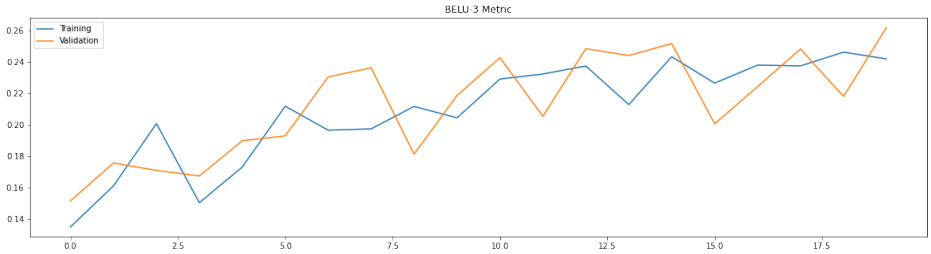
You can see BELU-1, 2, 3, and 4 metrics in Figure 17.



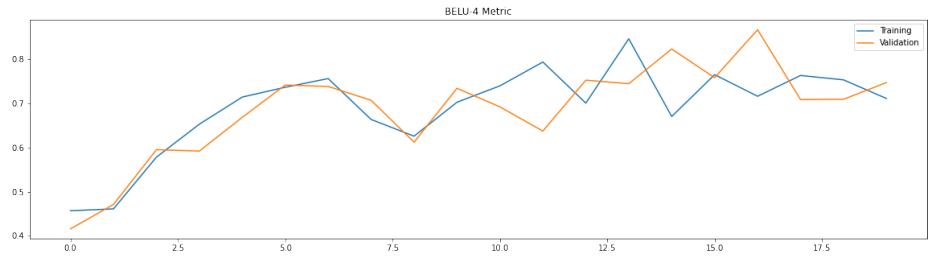
(a) BELU-1 Metric



(b) BELU-2 Metric



(c) BELU-3 Metric



(d) BELU-4 Metric

Figure 17: BELU Metrics (Higher is better)

5.2 Second Method

In this method, there is just object detector and depth estimator. by thresholding on the on the depth of each detected object, we split the detected objects to thee classes: Close, Middle, and Far.

5.2.1 Evaluation

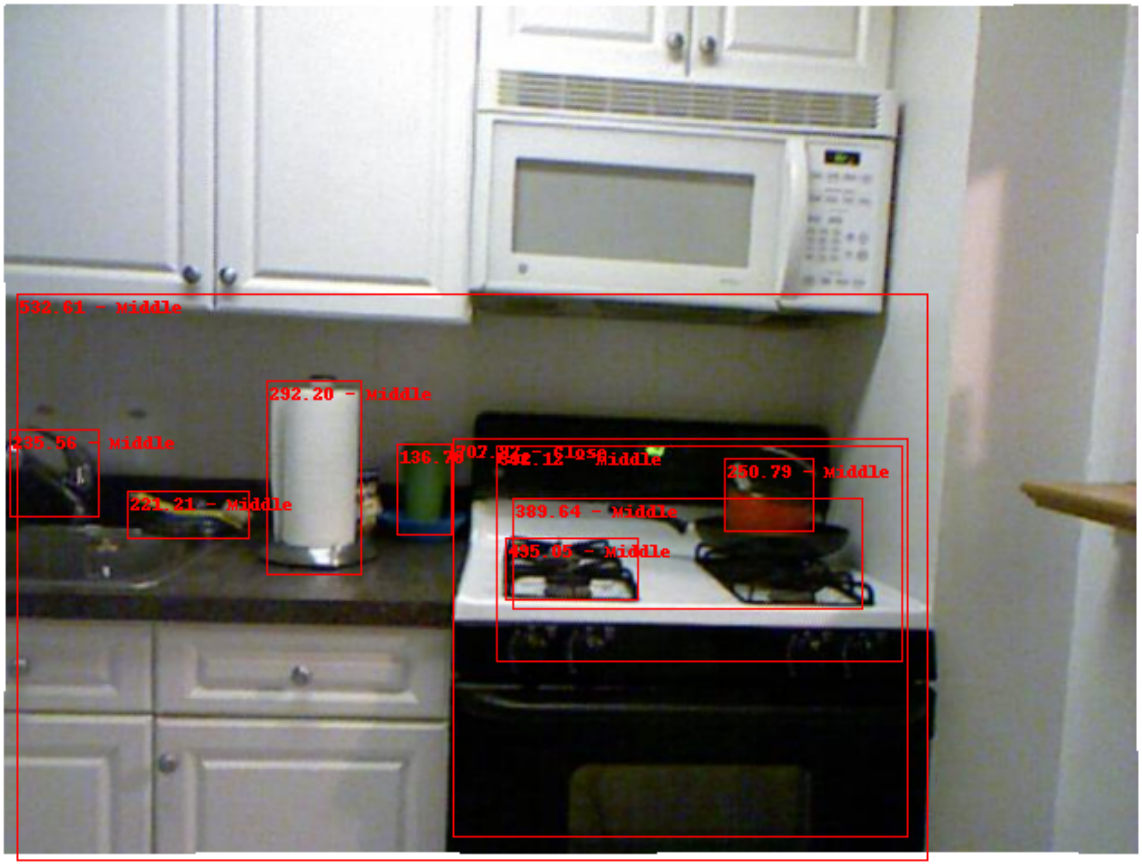
Because in this method we perform concatenation by thresholding on the depths, there is no numerical metric for measuring performance, but you can see the final outputs of the network in in Figures 18, 19, 20, and 21.

5.3 Test Images

You can see some test images in Figures 18, 19, 20, and 21.

(a) Detected Objects

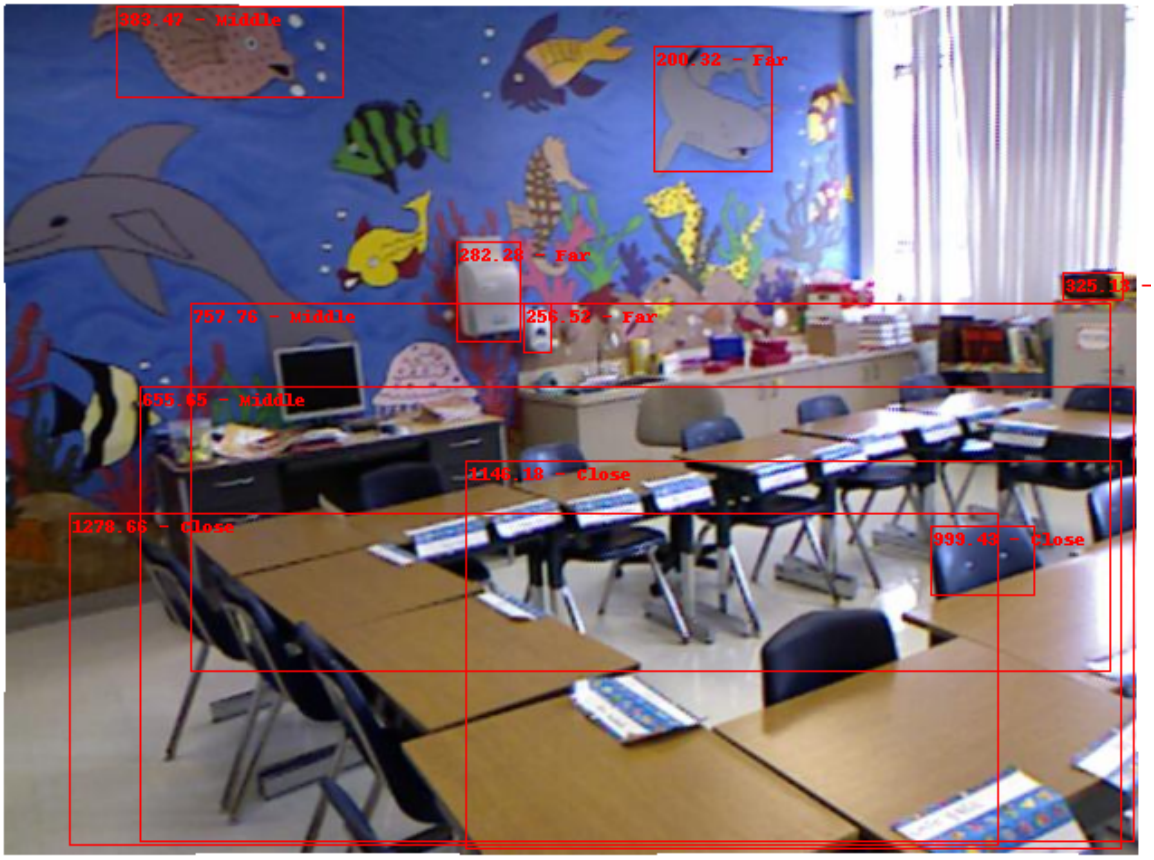
(b) Estimated Depths



(c) Final Output

Figure 18: Final Test Result 1

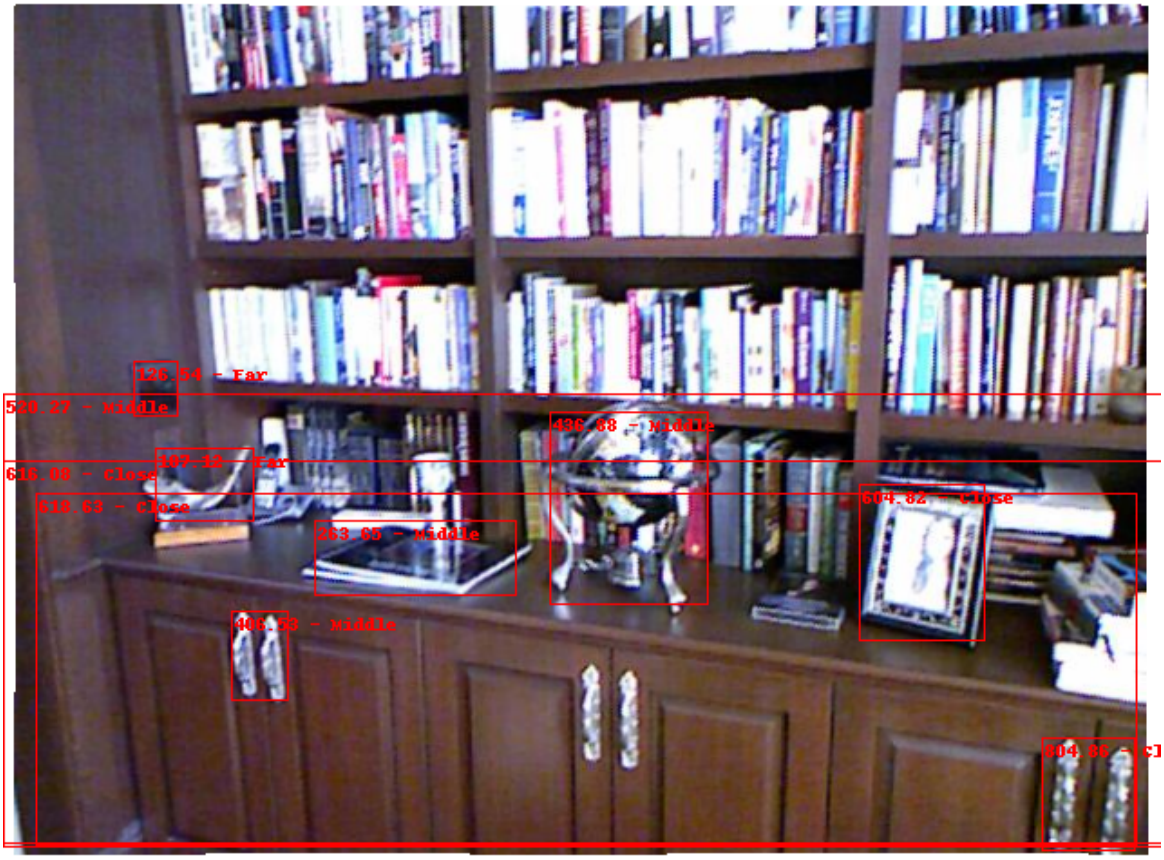
(a) Detected Objects (b) Estimated Depths



(c) Final Output

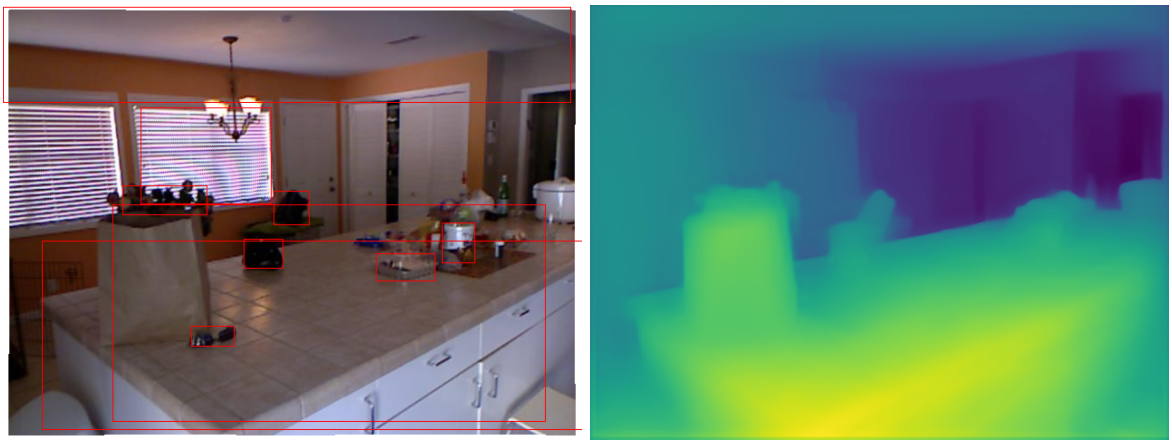
Figure 19: Final Test Result 2

(a) Detected Objects (b) Estimated Depths

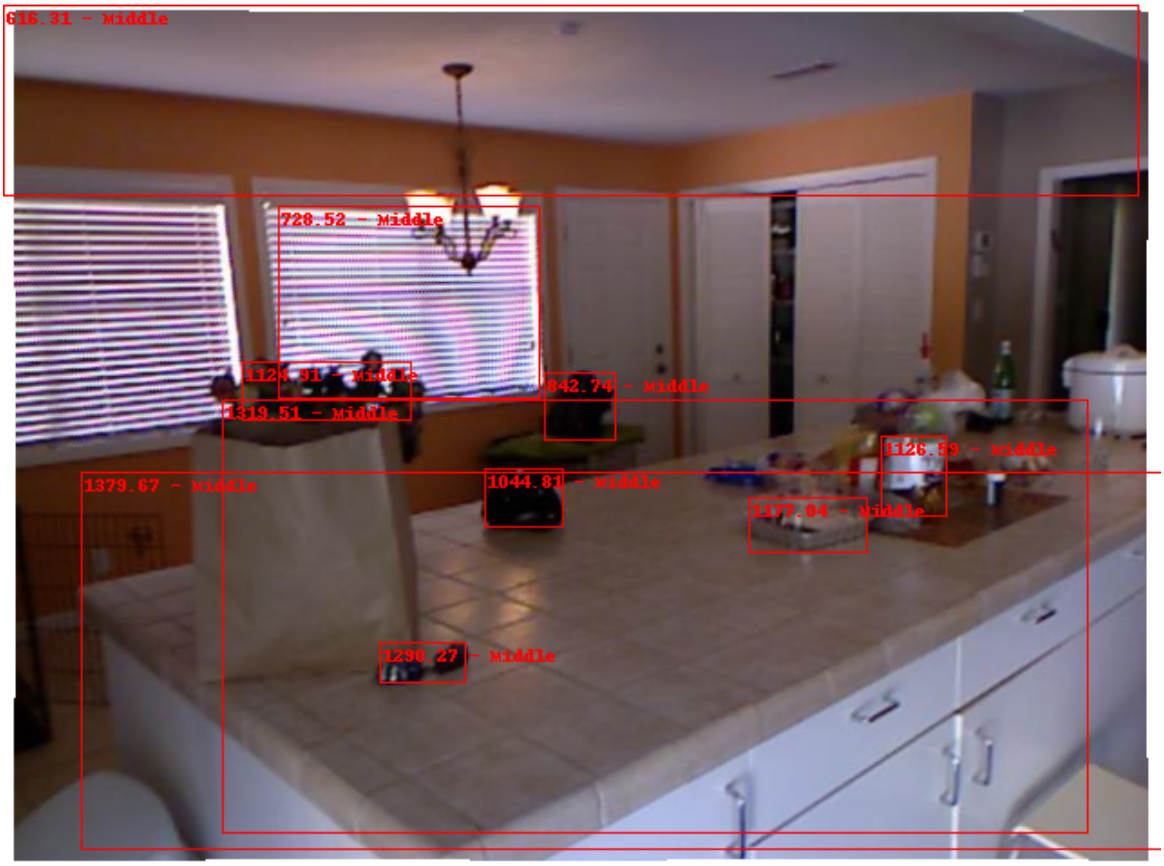


(c) Final Output

Figure 20: Final Test Result 3



(a) Detected Objects (b) Estimated Depths



(c) Final Output

Figure 21: Final Test Result 4

6 GUI

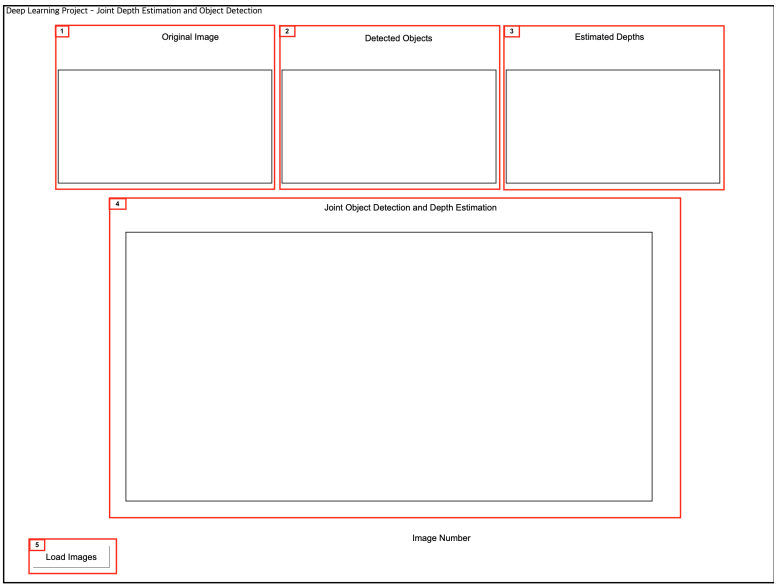
We designed a graphical user interface for using the network easily. You can load images to this app and then see the outputs of the network

6.1 Installation

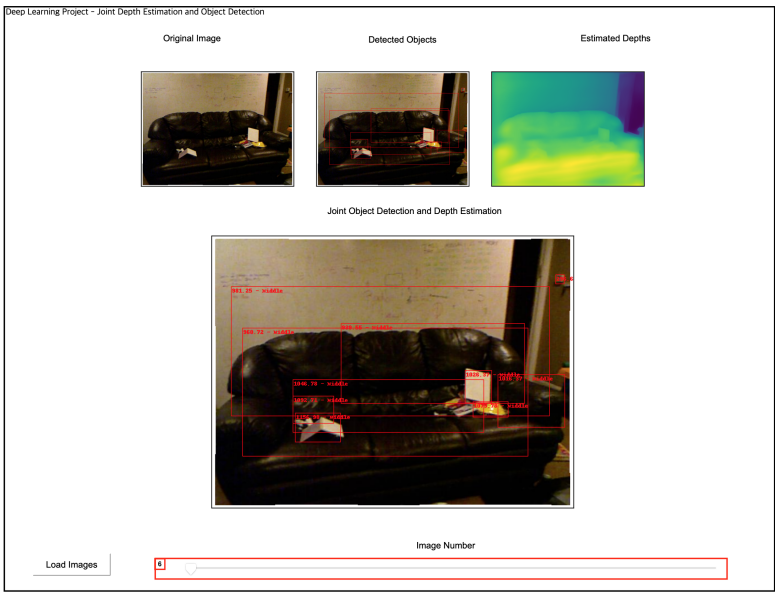
This app is placed in `./GUI` folder. You should first install dependencies and the run it using:

```
$ python3 GUI/main.py
```

6.2 App Enviornment



(a) GUI Before Loading any Image



(b) GUI After Loading Some Images

Figure 22: GUI Different Parts

You can see instructions for labeled parts in the Figure 22 in the Table 4.

Table 4: GUI Instruction Table

Label	Explanation
Label 1	This plot will show the original loaded image
Label 2	This plot will show the detected objects from the loaded image
Label 3	This plot will show the estimated depths of the loaded image
Label 4	This plot will show the final output of the loaded image
Label 5	This button is used for loading images
Label 6	By using this slider you can change the plotted image

It is noteworthy to mention that you can load more than 1 image using this app. Also, you should download the pretrained models before using the app.