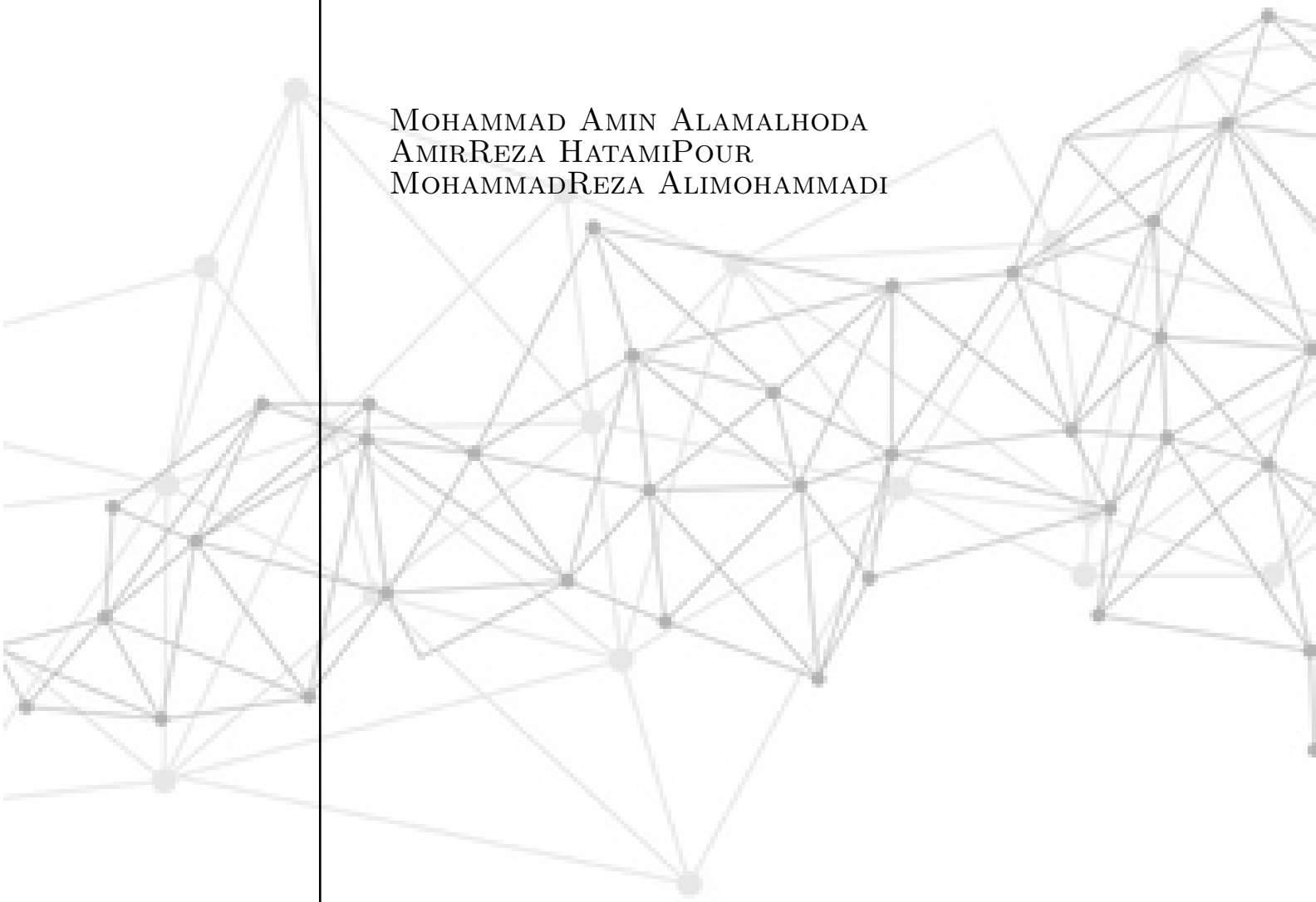
DL Course Project

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 makeinstall 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

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.

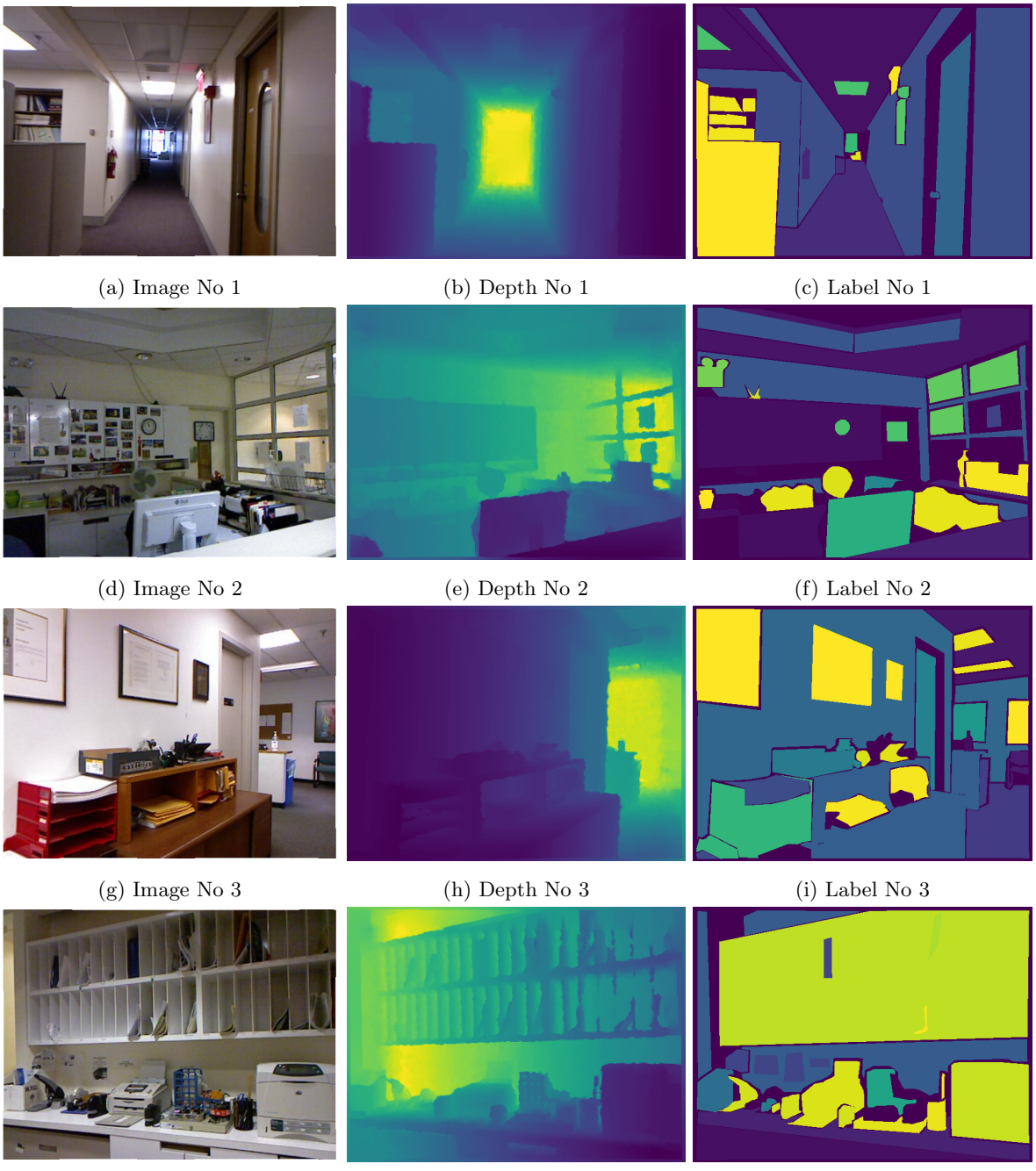
3 Datas

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

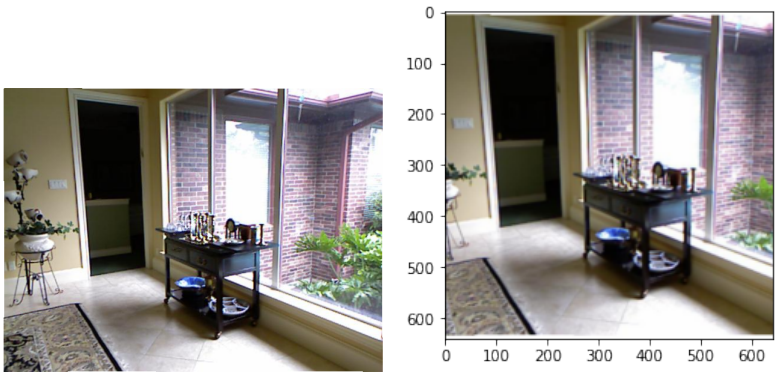
(j) Image No 4 (k) Depth No 4 (l) Label No 4

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.

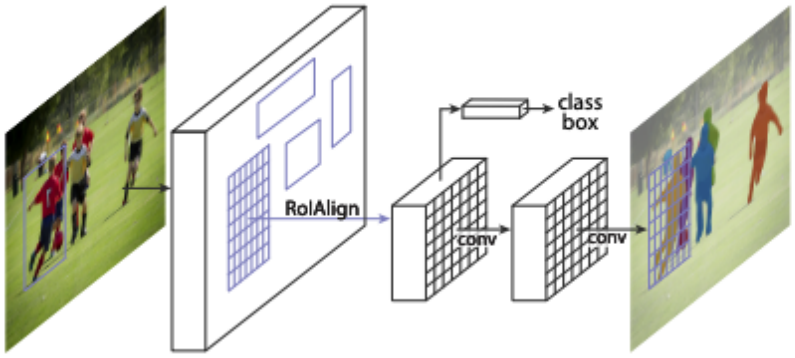


Figure 3: Bounding Box Regression in the Mask-RCNN

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 Pricedure

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

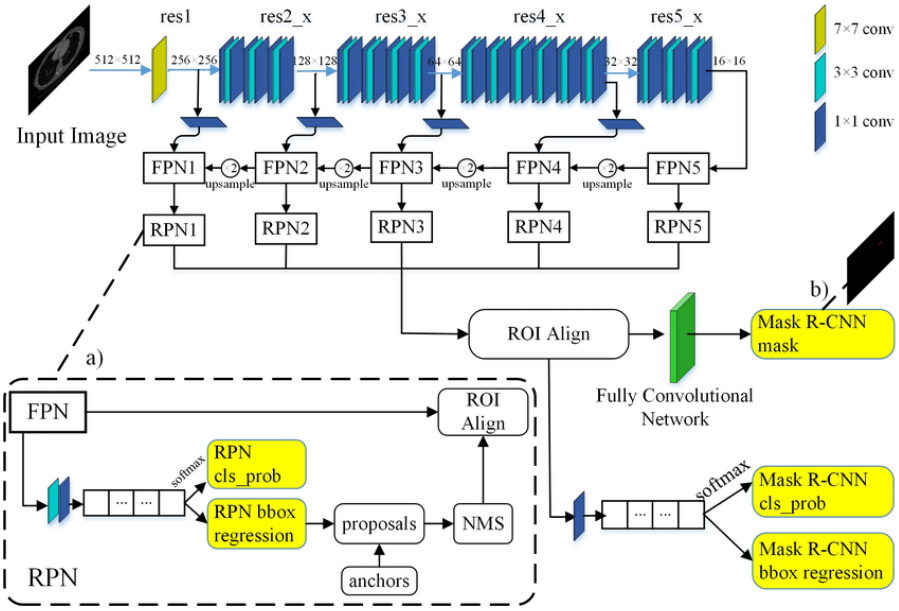


Figure 4: A Schematic of the Mask-RCNN

4.1.4 Fine-Tuning

As mentioned earlier, we fine tuned the model by using our datas. You can see the training loss plot in Figure 5.

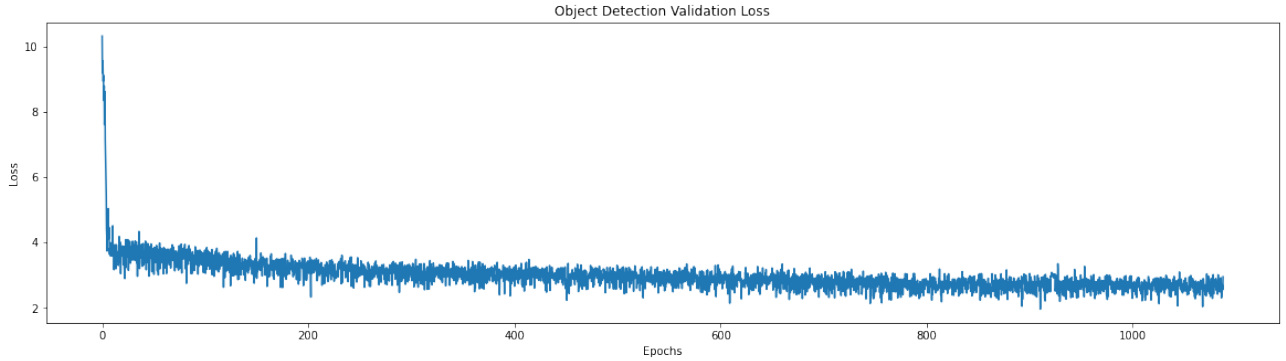


Figure 5: Training Loss of the Object Detector Model

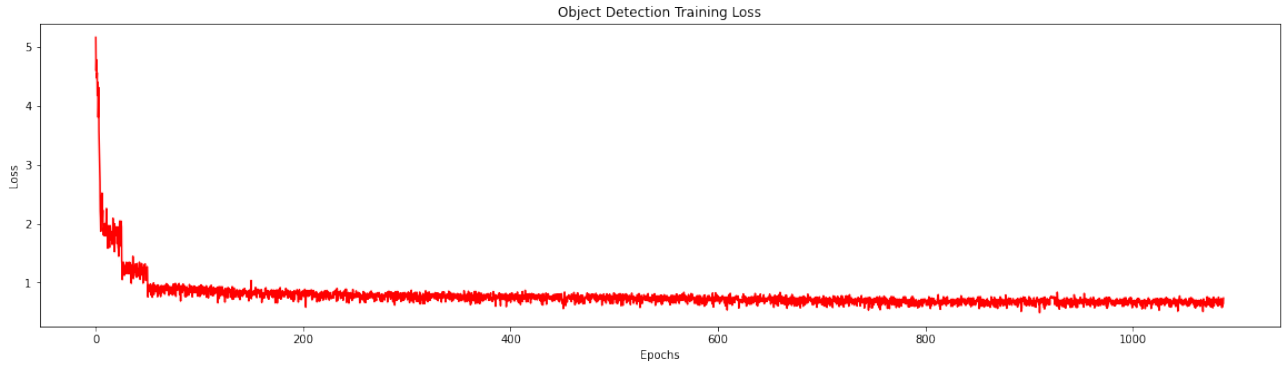


Figure 6: Validatoin Loss of the Object Detector Model

4.1.5 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

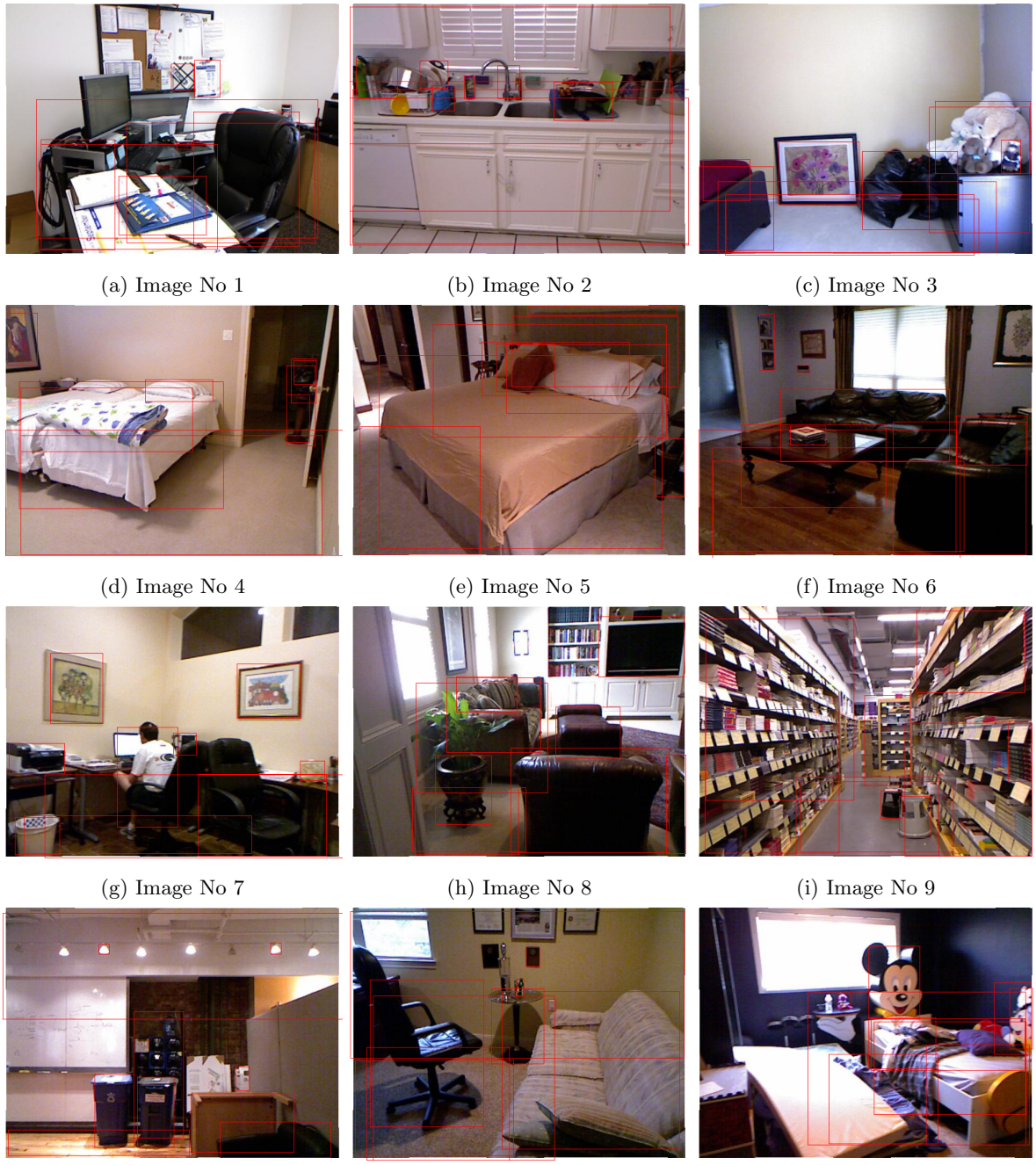
Therefore, we used different metrics for evaluating the object detector model which are written in Table 1:

Table 1: 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

4.1.6 Test Images

You can see some test images in the Figure 7.



(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 7: 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.

4.2.1 Architecture

4.2.2 Computation Pricedure

4.2.3 Fine-Tuning

As mentioned earlier, we fine tuned the model by using our datas. You can see the training loss plot in Figure 8.

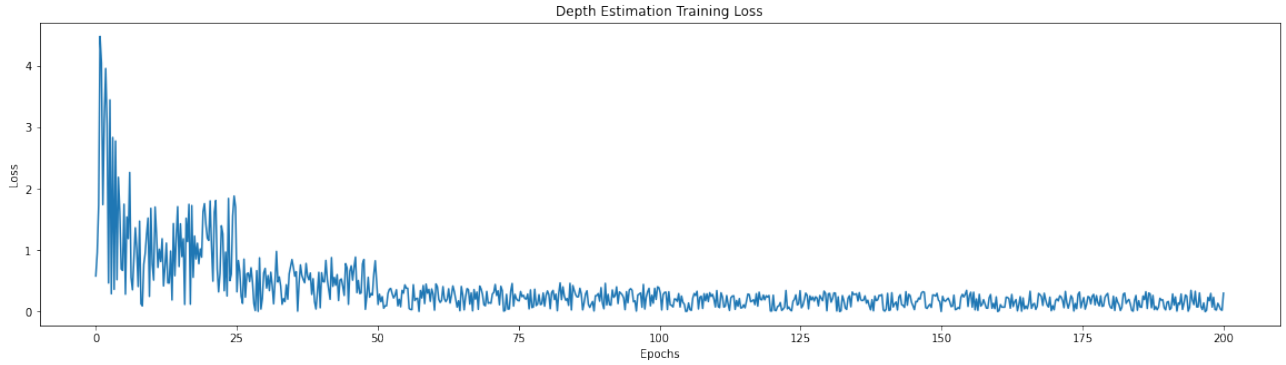


Figure 8: Training Loss of the Depth Estimator Model

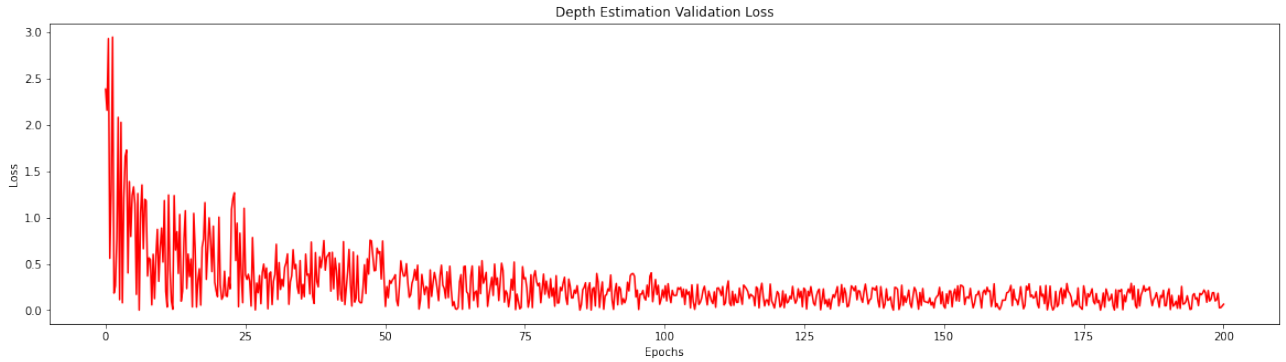


Figure 9: Vlidatoin Loss of the Depth Estimator Model

4.2.4 Evaluation

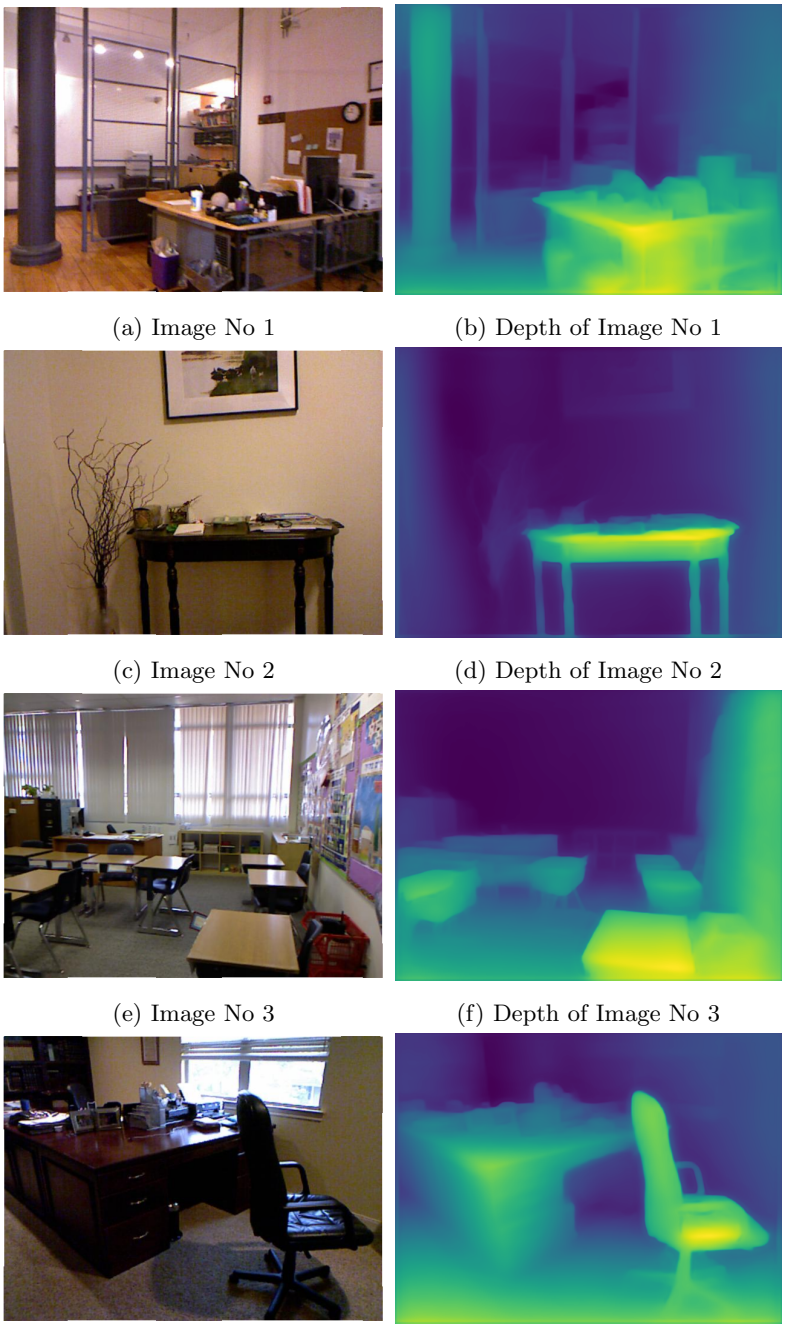
Estimating depth of different pixels in an image is a regression problem, so we used MSE(mean squad error) as loss. You can see this loss values for training and validation in the Table 2.

Table 2: 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

4.2.5 Test Images

You can see some test images in the Figure 10.



(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 10: Output of Depth Estimator Network for Some Test Images

5 Contacenator

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

5.0.1 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 11.

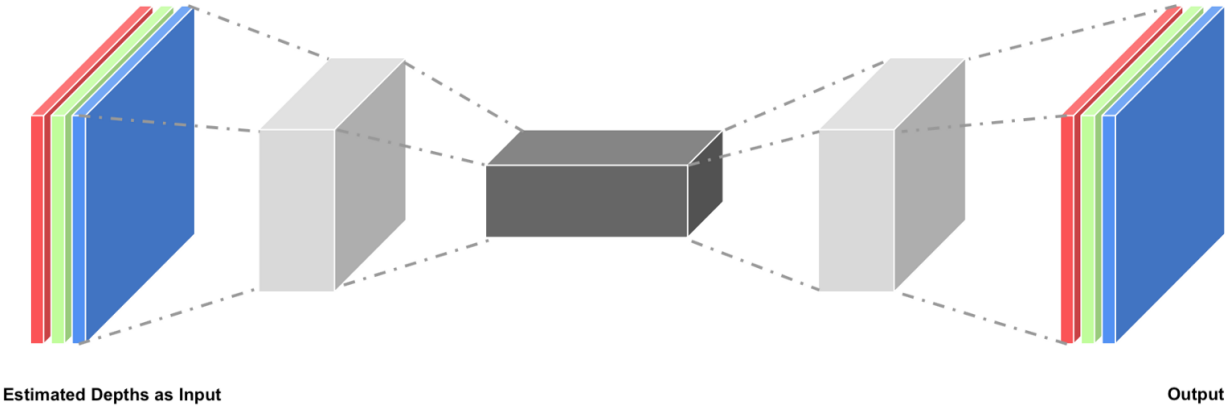


Figure 11: Schematic of the AE used for Concatenating

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.

5.0.2 Second Method

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