JACOBS
UNIVERSITY

Stereo Processing of Driving Images with MobileStereoNet

by

Michael Demse

Bachelor Thesis in Computer Science

Submission: May 14, 2022 Supervisor: Prof. Andreas Birk

Jacobs University Bremen | Department of Computer Science and Electrical Engineering

Statutory Declaration

Family Name, Given/First Name	Demse, Michael
Matriculation number	30003273
Kind of thesis submitted	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

14.05.2022

.....

Date, Signature

Abstract

Recent advancements in deep learning have shown that depth estimation from a stereo pair of images can be formulated by Convolutional Neural Networks (CNNs). However, most models proposed so far have failed to strike a good balance between accuracy and computational cost. To alleviate this problem, we leverage the lightweight stereo matching framework MobileStereoNet. MoblesStereoNets are based on MobileNet, an architecture specifically built for mobile and embedded vision applications. Conversion of disparity data to depth data is implemented by taking intrinsic and extrinsic properties into account as reprojection matrices are computed from parameters found in these properties. 3D models can be visualized either continuously by point clouds or discretely by voxel grids. The Gaussian mean and standard deviation of a point cloud density map are used to assess the quality of a point cloud. Autonomous vehicles make use of expensive and restricted depth sensors to build 3D awareness of their surroundings. The performance of these sensors declines in non-optimal conditions. This research aims to process and compare results from our proposed method with the aforementioned sensors and other various estimation methods.

Contents

1 Introduction

3D vision plays an important role in research and industries such as robotics, remote sensing, virtual reality, and industrial automation [36]. However, it has been difficult to reach satisfactory solutions with the previously used and traditional methods of 3D vision [16]. Because of this, 3D vision has been one of the most extensively investigated fields in computer vision and has garnered the attention of many researchers [35]. Multiple solutions and approaches have been explored; most of the popular ones being dedicated sensors and devices such as the LiDAR, RADAR, Microsoft Kinect, and various other structured light sensors. These sensors have been rigorously tried and tested in various scenarios and environments [23]. Through this, it was found that these methods provide accurate detection rates in regulated environments, however, their precision declines in non-optimal conditions such as poor lighting and sparse depth data [27]. These dedicated sensors and devices are usually expensive to acquire and maintain while needing high expertise to keep in operation [33]. Due to this, researchers have continued researching alternatives that are more flexible, adaptable, and cost-efficient.

Stereo vision takes aspiration from the human visual system as it achieves depth perception by mimicking binocular viewing points corresponding to each one of our eyes. The brain can create a complex 3D representation of our visual surroundings by leveraging the binocular disparities from a pair of 2D images created on the retinas. Binocular disparities are tiny positional discrepancies between similar features in the two retinal pictures caused by the horizontal separation of the two eyes. These differences are a rich source of information on 3D scene structure, and they are sufficient for depth perception on their own [20].

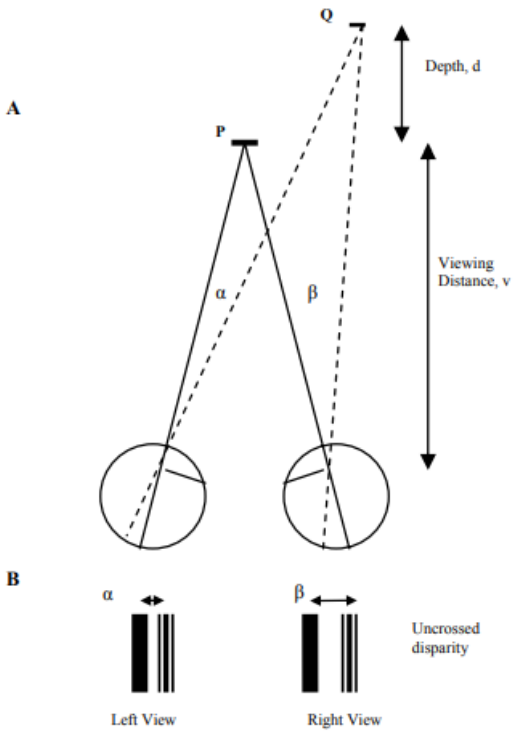


Figure 1: Stereo vision in the human visual system [20]

Disparity is the difference in placement of a point in the left and right views as viewed by the left and right eye. It is caused by parallax, horizontal separation of the eyes, and is used by the brain to calculate depth information from two-dimensional visuals. Without using dedicated sensors, we can use two ordinary calibrated and horizontally aligned 2D cameras, to estimate depth. This process is called stereo vision and the technique by which we do it is called stereo matching [22].

The cameras are displaced horizontally from one another to obtain a different view of the same scene.

(a) Left view (b) Right view

Figure 2: Stereo Images

Stereo matching is implemented by matching identical points in both scenes. These points will be found on the same horizontal line, further diminishing our search space, if our images are rectified. If our views are only horizontally different from each other, they are said to be rectified [15]. From this disparity and given some physical parameters of the camera, we can estimate the depth of points in our scenes. The goal of depth estimation is to recover, for every image pixel, the distance from the camera center to the nearest 3D scene point along the pixel's viewing direction [14].

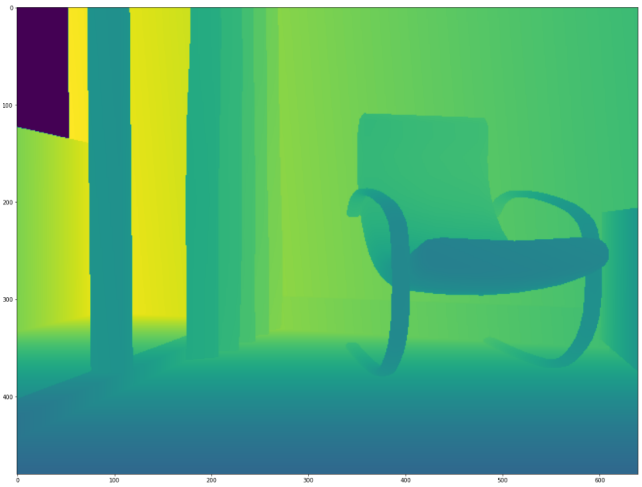
Stereo matching techniques can be subdivided into local and global techniques. One of the basic local techniques, Stereo Block Matching, works by taking a small window on the first scene and calculating the disparity between the same window position on the second scene [29]. Local methods are considered traditional due to limitations such as requiring a higher degree of binocular similarity and having high error rates [25].

Deep learning is a sub-category of machine learning that allows computational models with several processing layers to learn multiple degrees of abstraction for data representations. These techniques have vastly enhanced the state-of-the-art in speech recognition, visual object recognition, object detection, and a variety of other fields. Deep learning is mainly done through neural networks which aim to mimic the way that the human brain and neurons function. Neural networks contain multiple layers where each layer contains multiple nodes. Between different nodes of adjacent layers, weights and biases are calculated and assigned as a result of a process called backpropagation. Traditional neural networks are only suited for processing one-dimensional data. Convolutional Neural Networks on the other hand are built to handle data in the form of several arrays, such as a color image made up of three 2D arrays storing pixel intensities in each of the three color channels. Multiple arrays are used to represent many data modalities: 1D for signals and sequences, 2D for images or audio spectrograms; and 3D for video or volumetric imagery [18].

With the recent developments of deep learning and computer vision, the depth result from two stereoscopic cameras might be as precise as dedicated depth sensors while being more cost-efficient and flexible in different environments. As previously thought, the computational cost of making and running deep learning models is very high. In this thesis, we will explore the implementation, accuracy, and visual representation of one of the more recent lightweight deep learning and stereo matching framework known as MobileStereoNet [27].

The main purpose of the framework, MobileStereoNet, is to generate disparity images given a rectified pair of images. This however is not the final goal as we are still displaying a three-dimensional scene in two dimensions only. We need to find ways to translate this 2D data into a 3D model. This is where point clouds and voxel grids are of importance. A point cloud is a method of 3D presentation that preserves geometric information in 3D space without any discretization [11]. Voxel grids are derived from point clouds and are quantized, discrete fixed size point clouds [23]. The following figures show a sampled original image from the RedWood Dataset and the corresponding depth map, point cloud, and voxel grid representation.

(a) Original Image (Left View)



(b) Depth map

Figure 3: Sample Image and Depth map

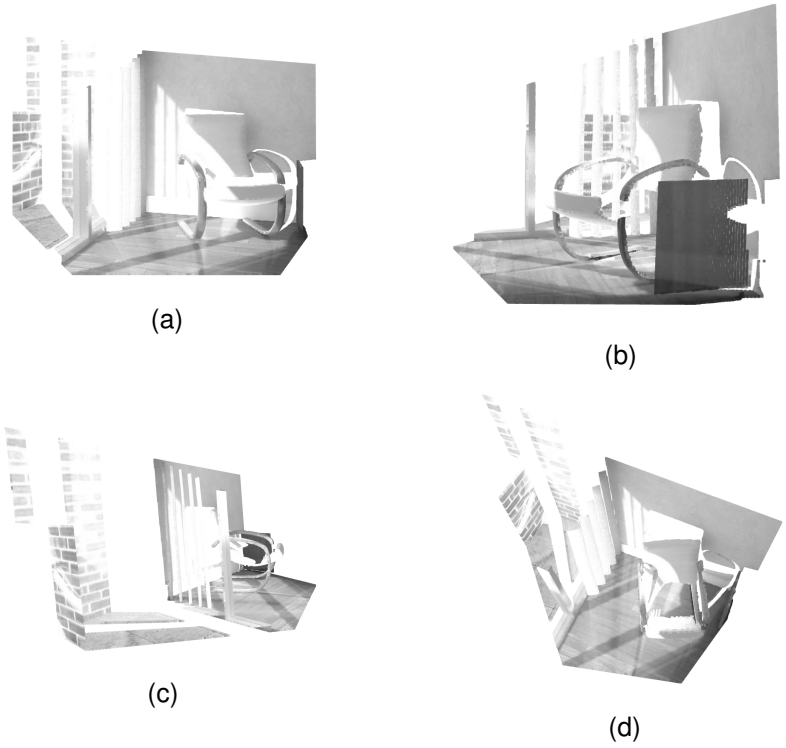


Figure 4: A point cloud viewed from different angles

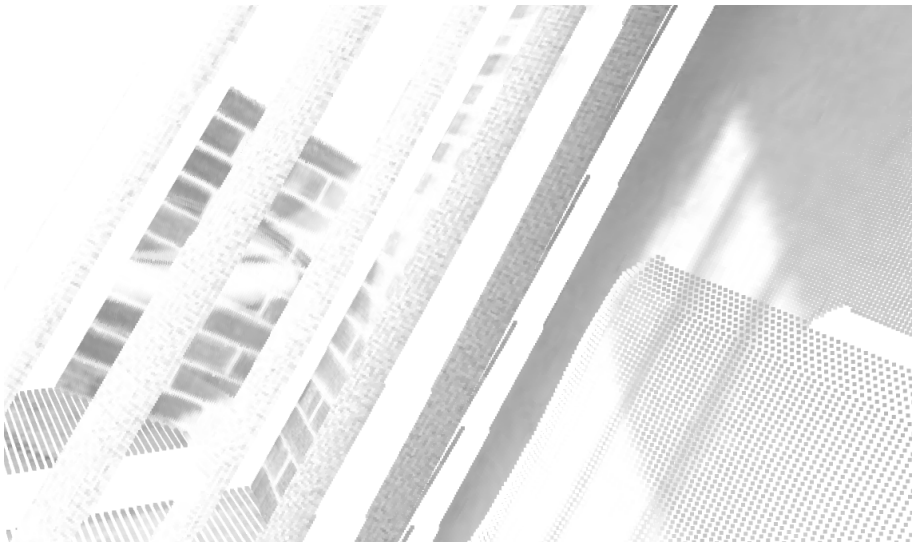


Figure 5: Point Cloud zoomed in

As you can see from Figure 5, a point cloud consists of points floating in space and is theoretically continuous. Voxel grids on the other hand eliminate this characteristic of point clouds and are a collection of discrete grids and not points.

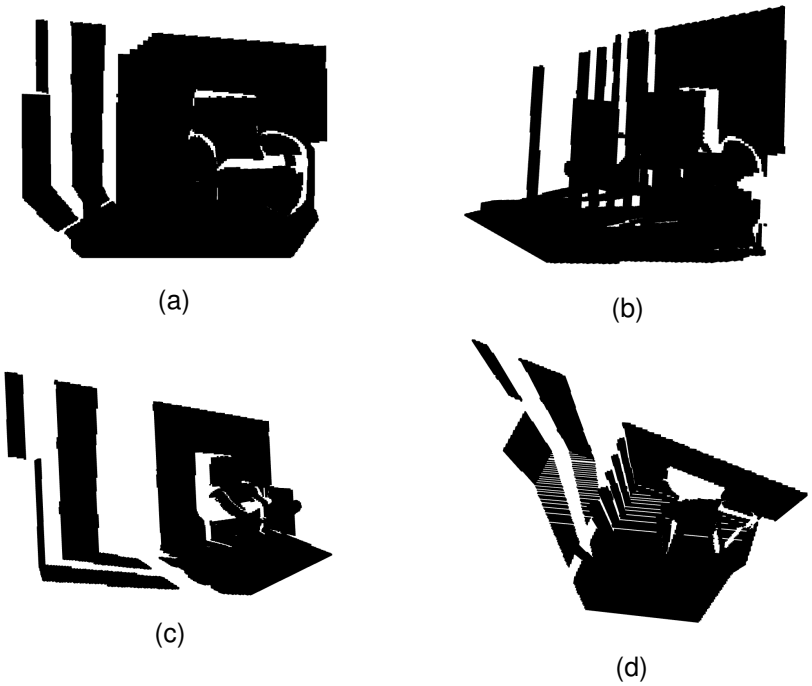


Figure 6: A Voxel Grid viewed from different angles

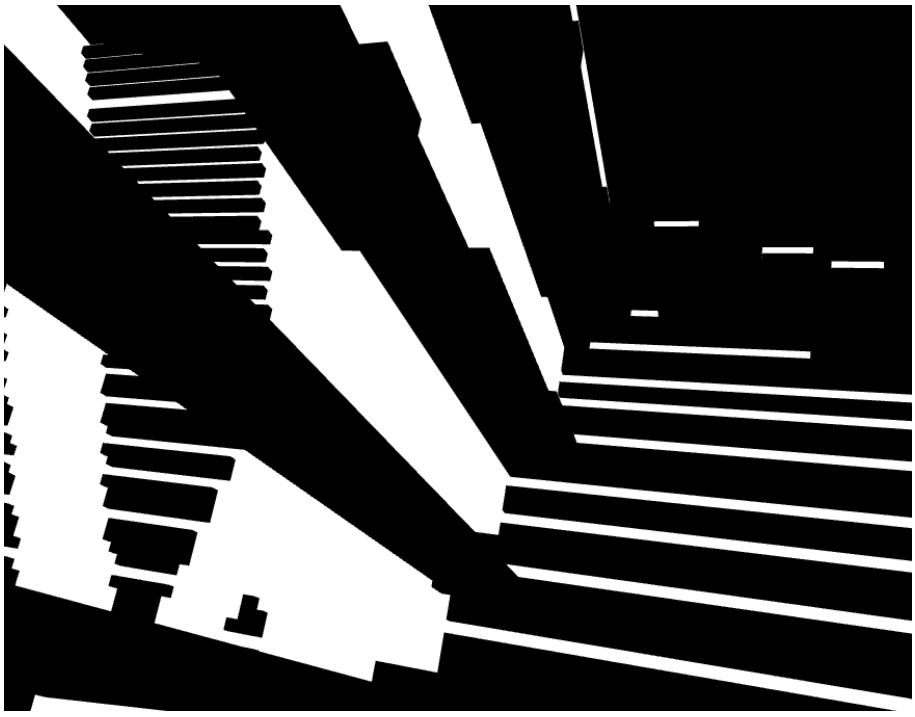


Figure 7: Voxel Grid zoomed in

2 Statement and Motivation of Research

The increase in autonomous driving has long been speculated to be the future of transportation and mobility as it offers not only convenience and comfort to passengers but also various other benefits such as reduction in traffic congestion and accidents [4]. The terms autonomous driving, autonomous vehicles, and self-driving vehicles are generally used to describe modes of transportation that require little or no human intervention. Like any piece of innovation, there have been multiple challenges associated with this. The most important and relevant has been increasing surrounding awareness based on 3D imagery. Because of this, autonomous driving and autonomous vehicles have been subject to a lot of research in computer vision and AI. This has resulted in a lot of progress as many autonomous vehicles and autonomous driving systems are ongoing testing for commercialized public use.

In autonomous driving, perceiving the surrounding has been an essential and huge challenge. In order for an AV (Autonomous Vehicle) to construct a complete awareness of its surrounding, it needs to have sensors and cameras on each of its sides, and then combine and stitch every view [1]. A combination of different sensors is frequently used to achieve this with a reasonable performance [19]. The 3D model constructed from the images is then used to compute other important information such as how far obstacles in the view are and in what direction they are moving. The AV further uses this information to calibrate itself and decide its next step.[21]

It is critical for an autonomous vehicle to be able to perceive and distinguish obstacles in its environment accurately and reliably. Many approaches based on stereo vision or 2D/3D sensor technologies have been presented in recent years for various application areas and scenarios. LiDAR is a technique for determining ranges or variable distances that involves using a laser to target an object or a surface and measuring the time it takes for the reflected light to return to the receiver [8]. LIDAR and RADAR are the primary perception systems used in today's research prototypes developed by major industry and academic players. They usually give the vehicle a very accurate full 360 view, making it more aware of the environment than a normal human driver. The disadvantage of these systems is the high cost of implementation. Moreover, Time of Flight (ToF) devices like the LIDAR are only effective up to a distance of around 5–7 meters and are far too sensitive to be used in outdoor conditions, especially in highly lit environments. Passive sensors such as cameras are more reliable and robust than ToF sensors because they can produce high-resolution disparity maps and are suitable for both indoor and outdoor contexts [12]. LIDAR also has shortcomings with balancing range and resolution [3]. As a result, using cameras and computer vision techniques to replace these systems is a viable option [1].

Several algorithms have been tried to perform stereo matching between two images. In order to do this, we need to compute or estimate the difference in distance or disparity between corresponding points in the images. The most basic ones would be the sum of absolute differences (SAD), and the sum of squared differences (SSD). These algorithms or techniques use a local approach by taking a window and by computing the sum of subtracted values to find the disparity between two images. Other common algorithms for window-based techniques include the normalized cross-correlation (NCC), rank transforms (RT), and census transforms (CT) [17]. All these methods have different drawbacks such as low accuracy and being prone to discrepancy when having textureless regions and occlusions. Since accuracy is highly required in autonomous driving and other real-

time stereo matching applications we have to look at other ways of getting better results [24]. New methods and techniques for solving this problem are developed every year and exhibit a trend toward improvement in accuracy and time consumption [12].

All of the traditional techniques for disparity estimation, however, have the same flaw: they only consider one visual signal at a time, rather than aggregating and weighting the information offered by all of them. This is important not only because we miss out on a lot of information, but also because many psychological studies show that this information contributes highly when humans are dealing with the same problem. Furthermore, there is another component that the above-mentioned methodologies completely overlook: image statistics. As humans, we create a variety of assumptions about the world we live in that are beneficial to us. These are based on the repeated patterns we encounter in our daily lives and are an important element of our visual processing [2]. With the recent developments in the fields of machine learning and deep learning, different models have been suggested and have yielded better results than the traditional and localized techniques. They have been found to be more accurate and also more flexible with respect to the conditions they can be applied under and their performances in those environments. This can be referenced from the various standardized precision measuring datasets like KITTI. We use Convolutional Neural Networks which are neural networks mostly used for image processing for this task. They are ideal for this task since they are an optimal way to catch the statistics, as the training process naturally picks up the factors that are relevant and calculates the relevancy or weight of each factor.

These CNNs take a lot of time to train, test, and finally predict models. Therefore an alternative lightweight framework, MobileStereoNet has recently been developed with the aim of solving this problem. This framework is mainly based on the building blocks of MobileNet, which are mainly geared towards mobile devices. This means that these building blocks are built for speed on lower ended devices. By basing our neural network on this model we can improve the duration needed for our model to train, test, and also predict. MobileStereoNet has been tested on different datasets including the very large and popular SceneFlow dataset. This time we are going to apply to a dataset of driving images and assess its performance after modifying the model to make it more suitable for the task at hand. We will also produce point cloud and voxel grid representations of our scenes based on the depth map we have estimated as well as benchmark its performance by conducting cost and run-time analysis.

3 Description of the Investigation

3.1 Description of framework and dataset used

In the previous sections, we have outlined several techniques that have been used to compute the disparity between two images along with their setbacks. We have seen that these traditional techniques of disparity estimation lack accuracy but also that their potential solutions, neural networks, need a lot of computational power and time to operate and process. Recognition tasks must be completed in a timely manner on computationally constrained platforms in many real-world applications such as robotics, augmented reality and more importantly in our case, autonomous driving [13]. Taking this into consideration, several scientists from Google created MobileNets in 2017 by implementing depthwise separable convolutions. MobileNets are a type of convolutional neural network designed for mobile and embedded vision applications. They are based on a streamlined architecture that uses depthwise separable convolutions to build lightweight deep neural networks that have low latency for mobile and embedded devices [13].

Several attempts of making lightweight neural networks have been made before the invention of MobileNets. However, most of them resulted in a high accuracy trade-off. MobileNets are different from their predecessors as they maintain high accuracy while having low latency. Therefore, MobileNets have been proven to be very efficient ways of making lightweight neural networks that can run on mobile devices and other devices with low computing power. Because of this many libraries and frameworks have incorporated MobileNets into their workings. Stereo Vision has also benefited from MobileNets with the creation of MobileStereoNet, created by a group of researchers from the University of Tübingen in late 2021. Due to the recency of MobileStereoNets, there has been very little research or experiments conducted on them. At the time of writing, the only scientific paper available about MobileStereoNets is the one written by the creators themselves. This has made it increasingly difficult to gather data about the specifics of this framework.

MobileStereoNet comes with two models, one being 2D-MobileStereoNet which is based on 2D convolutions and 3D cost volume, and the second one being 3D-MobileStereoNet which is based on 3D convolutions and 4D cost volume. While the 3D model provides higher accuracy results compare to the 2D model, it is computationally more expensive due to the more complex networks. As a result, even on a moderate GPU, some of the networks will raise an out-of-memory (OOM) error [27]. Therefore, given the limited graphics power we have available to us, we have chosen to do all training, validation, and testing in this project with 2D-MobileStereoNet.

We have chosen KITTI, a popular dataset in the computer vision, mobile robotics, and autonomous driving community to train, validate, and test our model. When we were looking for a driving dataset, KITTI seemed to be an obvious option as it provides diverse, real-world traffic circumstances with a variety of static and dynamic items. More importantly, it provides the corrected and raw image sequences, as well as calibrated, synced, and timestamped data. The KITTI dataset provides grayscale and color stereo images from its four cameras, and the ground truth depth data from its Velodyne HDL-64E rotating 3D laser scanner [9]. All these characteristics and qualities make the KITTI dataset a perfect candidate for our investigation.

(a) Left View

(b) Right View

(c) Ground Truth Depth

Figure 8: Sample Image set in KITTI

3.2 Generating Disparity Map

MobileStereoNet comes with a model that is pre-trained on the SceneFlow Dataset. We can already use this model to generate disparity predictions for our KITTI dataset. However, this wouldn't yield very accurate results as the objects in the SceneFlow dataset are substantially different from the ones in KITTI. To alleviate this problem, we have two options. One is training the whole network from scratch with the KITTI dataset, and the other is fine-tuning our pre-trained model to our current dataset. Fine-tuning is a transfer learning technique in which knowledge gained from training one type of problem is applied to another similar task or domain [31]. This is usually done by removing and replacing the last few layers of the pre-trained model. Training our model from scratch is not an option as it requires a large amount of training data and also is computationally heavy. Fine-tuned CNNs perform as well as fully trained CNNs and sometimes outperform them when limited training data is available [28].

After dividing the KITTI dataset into testing and training sections, we have moved them to the `MobileStereoNet/datasets` folder. The next step would be generating the training, validation, and testing lists for the images located in these directories. The following diagram shows our resulting folder structure.

```
MobileStereoNet
|- checkpoints
|--|- MSNet2D_SF.ckpt
|--|- MSNet3D_SF.ckpt
|- datasets
|--|- kitti_data_scene_flow
|--|--|- testing
|--|--|--|- image_2
|--|--|--|- image_3
|--|--|- training
|--|--|--|- disp_noc_0
|--|--|--|- image_2
|--|--|--|- image_3
|- filenames
|--|- kitti_test.txt
|--|- kitti_train.txt
|--|- kitti_val.txt
```

The training list contains the location of the images from which the model learns patterns in the data. The same training data is fed to the neural network architecture repeatedly during each epoch, and the model continues to learn the data's features. The validation list routes to the data that is separate from the training set and is used to test the performance of our model during training. This validation process provides data that we can use to fine-tune the model's hyperparameters and configurations. We have arbitrarily chosen the training validation split to be the classic 80 20 split. The following are our training and validation lists respectively. As you can see our training list takes two images (left and right), corresponding to the image_2 and image_3 folders.

Training list: kitti_train.txt

```
training/image_2/000179_10.png training/image_3/000179_10.png
training/disp_occ_0/000179_10.png
training/image_2/000128_10.png training/image_3/000128_10.png
training/disp_occ_0/000128_10.png
training/image_2/000122_10.png training/image_3/000122_10.png
training/disp_occ_0/000122_10.png
training/image_2/000178_10.png training/image_3/000178_10.png
training/disp_occ_0/000178_10.png
.
.
.
```

Validation list: kitti_val.txt

```
training/image_2/000160_10.png training/image_3/000160_10.png
training/disp_occ_0/000160_10.png
training/image_2/000009_10.png training/image_3/000009_10.png
training/disp_occ_0/000009_10.png
training/image_2/000137_10.png training/image_3/000137_10.png
training/disp_occ_0/000137_10.png
training/image_2/000087_10.png training/image_3/000087_10.png
training/disp_occ_0/000087_10.png
```

.
.
.

Once we have made the training and validation lists we run `train.py` with the following arguments to fine-tune our model. The epochs and batch size can be adapted according to the computational power of the system. This experiment is conducted on a NVIDIA Tesla P100 graphics card with 16GB of memory. Our moderate graphics card allows us to assign the maximum assigned batch size of 2 without raising an out-of-memory error. We have a relatively high number of epochs in order to achieve high accuracy on our model. However, we have to consider the chance of over-fitting and over-maximizing the number of epochs simply because our hardware can handle it. In this example, we have decided to have 400 epochs. An epoch is a unit of time used to train a neural network with all of the training data for a single cycle. We use all of the data exactly once in an epoch. A forward and backward pass are combined to make one pass. An epoch is made up of one or more batches in which we train the neural network using a portion of the dataset. Iteration is the term used to describe the process of going through all of the training examples in a batch [5].

```
python train.py --dataset kitti
--datapath "./datasets/kitti_data_scene_flow"
--trainlist ./filenames/kitti15_train.txt
--testlist ./filenames/kitti15_val.txt
--epochs 400 --lrepochs "200:10" --batch_size 2 --test_batch_size 2
--loadckpt ./checkpoints/MSNet2D_SF.ckpt --model MSNet2D --logdir "./checkpoints"
```

Running the above command starts the fine-tuning process.

```
Loading model ./checkpoints/MSNet2D_SF.ckpt
Start at epoch 0
Downscale learning rate at epochs: [200], downscale rate: 10.0
Setting learning rate to 0.001
Epoch 0/400, Iter 0/79, train loss = 4.264, time = 9.751
Epoch 0/400, Iter 1/79, train loss = 4.051, time = 0.692
Epoch 0/400, Iter 2/79, train loss = 3.026, time = 0.663
Epoch 0/400, Iter 3/79, train loss = 2.929, time = 0.670
Epoch 0/400, Iter 4/79, train loss = 2.597, time = 0.667
Epoch 0/400, Iter 5/79, train loss = 3.545, time = 0.661
Epoch 0/400, Iter 6/79, train loss = 4.498, time = 0.663
Epoch 0/400, Iter 7/79, train loss = 4.286, time = 0.660
Epoch 0/400, Iter 8/79, train loss = 2.507, time = 0.663
Epoch 0/400, Iter 9/79, train loss = 3.246, time = 0.662
Epoch 0/400, Iter 10/79, train loss = 1.957, time = 0.666
```

Figure 9: Epoch 0 in the fine-tuning process

After every epoch our process provide values for loss and the thresholds.

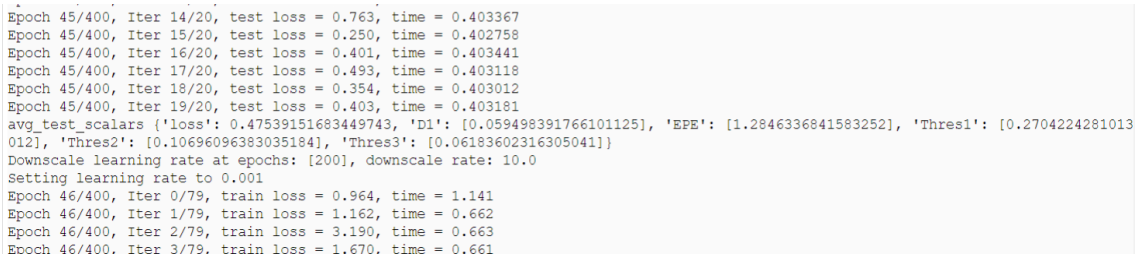


Figure 10: End of Epoch results

After the fine-tuning process is over, our new fine-tuned model will be created in the checkpoints folder. The checkpoints in between will also be saved in the folder to track progress and version history.

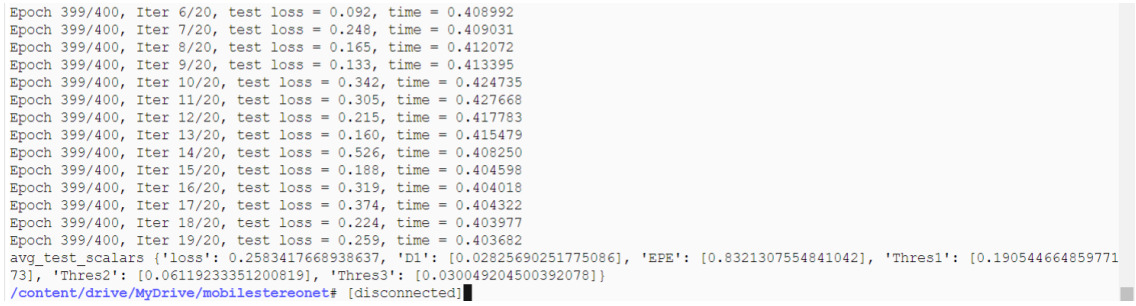


Figure 11: At the end of the fine-tuning process

Now that we have our fine-tuned model, we can finally go ahead and make our prediction for our images in the testing folder. Before we proceed we need to make a test list of the images we want to predict the disparity for.

Testing List: kitti_15.txt

```
testing/image_2/000000_10.png testing/image_3/000000_10.png
testing/image_2/000001_10.png testing/image_3/000001_10.png
testing/image_2/000002_10.png testing/image_3/000002_10.png
testing/image_2/000003_10.png testing/image_3/000003_10.png
```

Once we have the test list, we can proceed to do the predictions. MobileStereoNet comes with a very intuitive `prediction.py` file. Providing the appropriate checkpoints, model and testing list, our command will be as follows.

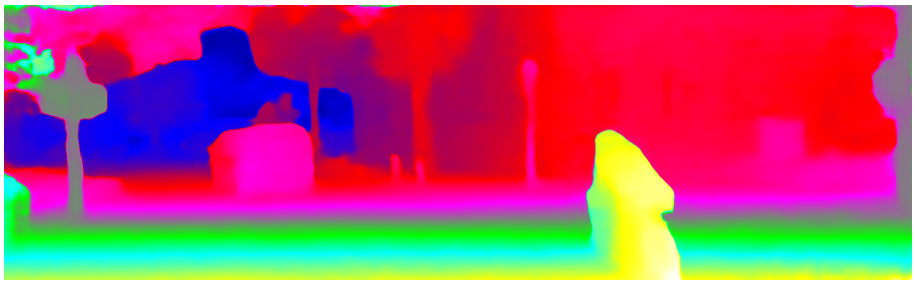
```
python prediction.py --datapath "./datasets/kitti_data_scene_flow"
--testlist ./filenames/kitti15_test.txt
--loadckpt ./checkpoints/finetuned.ckpt
--dataset kitti --colored 1 --model MSNet2D
```

When this command is run, it creates a folder named `predictions` and dumps all the predicted disparity images into the folder. On execution, it gives the following confirmation.

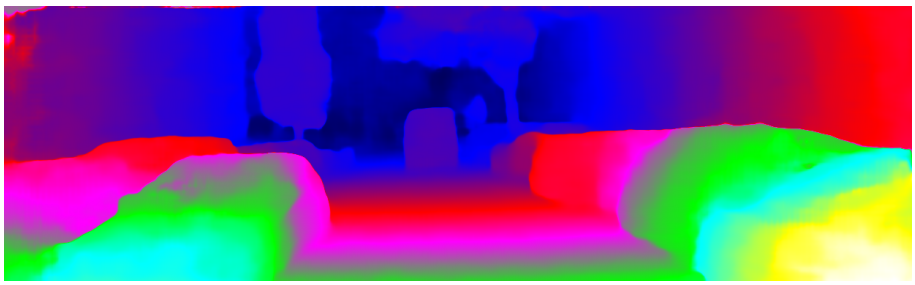
```
/content/drive/MyDrive/mobilestereonet# python prediction.py --datapath "./datasets/kitti_data_sce
ne_flow" --testlist ./filenames/kitti15_test.txt --loadckpt ./checkpoints/finetuned\ from\ repo.ck
pt --dataset kitti --colored 1 --model MSNet2D
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:490: UserWarning: This DataL
oader will create 4 worker processes in total. Our suggested max number of worker in current syste
m is 2, which is smaller than what this DataLoader is going to create. Please be aware that excess
ive worker creation might get DataLoader running slow or even freeze, lower the worker number to a
void potential slowness/freeze if necessary.
  cpuset_checked))
Loading model ./checkpoints/finetuned from repo.ckpt
Generating the disparity maps...
Done!
/content/drive/MyDrive/mobilestereonet#
```

Figure 12: After prediction of disparity images

Here are two sample disparity maps predicted in the `--colored 1` mode. There also exists a grayscale mode when `--colored 0` is set. As you can see colors are assigned to the pixels based on their disparity size between the views and therefore indicating how far the points are from the viewing point. In grayscale mode closer points are lighter while farther points are darker.



(a) Disparity map for Figure 8



(b)

Figure 13: Sample predicted disparity maps of KITTI on MobileStereoNet

3.3 Generating Point Clouds

Now that we have predicted the disparity map for our images, we have potential depth information in the form of 2D data. The next step would be to create a 3D presentation of our data. In order to convert our disparity data to depth data and then to a point cloud, we need to find different parameters of our camera which guide our camera in translating the real-world 3D data to 2D. These parameters are normally represented with matrices for a camera, representing each its properties. Most computer vision datasets come with this data as it is essential to understand and estimate camera properties. The KITTI dataset comes with the `calib_cam_to_cam` folder which includes different calibration files for each scene. For the sample stereo images given in Figure 8, this is the `calib.txt` provided.

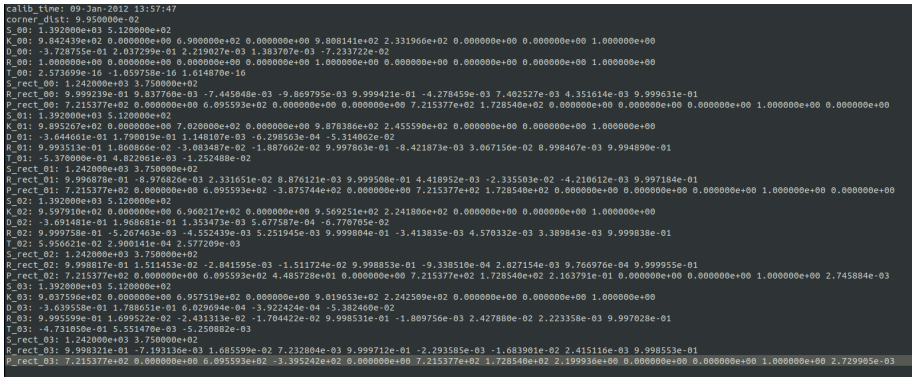


Figure 14: Calibration File for Figure 8

As mentioned before the KITTI dataset is collected from 4 different cameras and this file provides all calibration parameters for each one of them. In Figure 14 every matrix is followed by a two-digit number index representing the camera index. The matrices are given in a plain, unformatted text format which makes it difficult to extract the needed parameters. Therefore in order to visualize and interpret the matrices, we need to formalize our calibration file. A quick look at the KITTI documentation[9] provides us with the following definitions.

$$
\begin{aligned}
s^{(i)} \in \mathbf{N}^2 &\text{ ..... original image size(1392}\times\text{512) pixels} \\
K^{(i)} \in \mathbf{R}^{3\times3} &\text{ .... calibration matrices (unrectified)} \\
d^{(i)} \in \mathbf{R}^{5} &\text{.....distortion coefficients (unrectified)} \\
R^{(i)} \in \mathbf{R}^{3\times3} &\text{.....rotation from camera 0 to camera i} \\
t^{(i)} \in \mathbf{R}^{1\times3} &\text{.....translation from camera 0 to camera i} \\
s^{(i)}_{rect} \in \mathbf{N}^2 &\text{.....image size after rectification} \\
R^{(i)}_{rect} \in \mathbf{R}^{3\times3} &\text{.....rectifying rotation matrix} \\
P^{(i)}_{rect} \in \mathbf{R}^{3\times4} &\text{.....projection matrix after rectification}
\end{aligned}
$$

The process by which we estimate the parameters of a lens or image sensor of a camera is called camera calibration or camera resectioning. The parameters can be then used to establish the location of the camera in the picture or measure the size of an object in world units [6]. Camera Calibration is therefore used to find the internal or intrinsic parameters of the cameras such as the focal lengths and principal points, while camera localization aims to establish the pose of the camera in the world coordinate system [34]. Provided this information, we can start to make sense of our calibration file. Visualizing our matrices in a more formalized way gives us this result. Cameras 0 and 1 correspond to the left and right grayscale cameras while Cameras 2 and 3 correspond to the left and right color cameras.

```
Camera 0
S_00:  [1392.  512.]
K_00:  [[984.2439   0.     690.    ]
 [  0.     980.8141 233.1966]
 [  0.       0.       1.    ]]
D_00:  [-0.3728755   0.2037299   0.00221903  0.00138371 -0.07233722]
R_00:  [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
T_00:  [ 2.573699e-16 -1.059758e-16  1.614870e-16]
S_rect_00:  [1242.  375.]
R_rect_00:  [[ 0.9999239   0.00983776 -0.00744505]
 [-0.0098698   0.9999421  -0.00427846]
 [ 0.00740253  0.00435161  0.9999631 ]]
P_rect_00:  [[721.5377   0.     609.5593   0.    ]
 [  0.     721.5377 172.854    0.    ]
 [  0.       0.       1.       0.    ]]

Camera 1
S_01:  [1392.  512.]
K_01:  [[989.5267   0.     702.    ]
 [  0.     987.8386 245.559 ]
 [  0.       0.       1.    ]]
D_01:  [-0.3644661   0.1790019   0.00114811 -0.00062986 -0.05314062]
R_01:  [[ 0.9993513   0.01860866 -0.03083487]
 [-0.01887662  0.9997863  -0.00842187]
 [ 0.03067156  0.00899847  0.999489  ]]
T_01:  [-0.537       0.00482206 -0.01252488]
S_rect_01:  [1242.  375.]
R_rect_01:  [[ 0.9996878  -0.00897683  0.02331651]
 [ 0.00887612  0.9999508   0.00441895]
 [-0.02335503 -0.00421061  0.9997184 ]]
P_rect_01:  [[ 721.5377    0.      609.5593 -387.5744]
 [   0.      721.5377  172.854     0.    ]
 [   0.        0.        1.        0.    ]]

Camera 2
S_02:  [1392.  512.]
K_02:  [[959.791    0.     696.0217]
 [  0.     956.9251 224.1806]
 [  0.       0.       1.    ]]
D_02:  [-0.3691481   0.1968681   0.00135347  0.00056776 -0.06770705]
R_02:  [[ 0.9999758  -0.00526746 -0.00455244]
 [ 0.00525195  0.9999804  -0.00341384]
 [ 0.00457033  0.00338984  0.9999838 ]]
T_02:  [0.05956621 0.00029001 0.00257721]
S_rect_02:  [1242.  375.]
R_rect_02:  [[ 9.998817e-01  1.511453e-02 -2.841595e-03]
 [-1.511724e-02  9.998853e-01 -9.338510e-04]
 [ 2.827154e-03  9.766976e-04  9.999955e-01]]
P_rect_02:  [[7.215377e+02 0.000000e+00 6.095593e+02 4.485728e+01]
 [0.000000e+00 7.215377e+02 1.728540e+02 2.163791e-01]
 [0.000000e+00 0.000000e+00 1.000000e+00 2.745884e-03]]

Camera 3
S_03:  [1392.  512.]
K_03:  [[903.7596   0.     695.7519]
 [  0.     901.9653 224.2509]
 [  0.       0.       1.    ]]
D_03:  [-0.3639558   0.1788651   0.00060297 -0.00039224 -0.0538246 ]
R_03:  [[ 0.9995599   0.01699522 -0.02431313]
 [-0.01704422  0.9998531  -0.00180976]
 [ 0.0242788   0.00222336  0.9997028 ]]
T_03:  [-0.473105    0.00555147 -0.00525088]
S_rect_03:  [1242.  375.]
R_rect_03:  [[ 0.9998321  -0.00719314  0.01685599]
 [ 0.0072328   0.9999712  -0.00229358]
 [-0.01683901  0.00241512  0.9998553 ]]
P_rect_03:  [[ 7.215377e+02  0.000000e+00  6.095593e+02 -3.395242e+02]
 [ 0.000000e+00  7.215377e+02  1.728540e+02  2.199936e+00]
 [ 0.000000e+00  0.000000e+00  1.000000e+00  2.729905e-03]]
```

Figure 15: Formalised calibration matrices

These calibration matrices provide us with the parameters needed to convert our disparity into real-world depth data. The OpenCV library provides a very useful and simple function `cv2.reprojectImageTo3D(disp,Q)` to reproject 2D data to 3D. This function takes

two parameters, the first one being the disparity map and the second being the Q transformation matrix. Q is a 4x4 perspective, disparity-to-depth mapping transformation matrix that can be obtained with `cv2.stereoRectify` by taking our camera matrices as input. We have already predicted the disparity, however, we need to discover the intrinsic properties of our camera as input for computing Q. Using `cv2.stereoRectify`, we can find our transformation matrix Q along with other relevant computed matrices.

```
R1, R2, P1, P2, Q, roi_left, roi_right = cv2.stereoRectify(K_00,
D_00, K_01, D_01, S_00, R_01, T_01, alpha=1.0)
```

Plugging our matrices in we get the following Q matrix for our sample images from Figure 8:

$$Q = \begin{bmatrix} 1 & 0 & 0 & -684.09642029 \\ 0 & 1 & 0 & -235.54417801 \\ 0 & 0 & 0 & 728.62628147 \\ 0 & 0 & 1.86161607 & 0 \end{bmatrix}$$

Another way of calculating Q is by estimating each element of the matrix based on the camera parameters. This method provides more understanding of how the Q matrix is estimated. The camera intrinsic matrix K, is defined as [10] $\begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$ where f_x is focal length in the x direction, f_y is focal length in the y direction, c_x and c_y are the position of the principal point and s is shear.

We can see that our intrinsic matrix for camera 0 from Figure 15 is given by $K_{00} = \begin{bmatrix} 984.2439 & 0 & 690 \\ 0 & 980.8141 & 233.1966 \\ 0 & 0 & 1 \end{bmatrix}$. Correlating these matrices we can simply see that, for camera 0:
$f_x = 984.2439$, $f_y = 980.8141$, $c_x = 690$, $c_y = 233.1966$.
Shear (s) is always equal to 0 as we are using a non-analog camera.

```
import cv2
# Camera 0 or left camera is world origin
C0 = np.zeros(3)
C1 = -np.linalg.inv(P_rect_01[:,:3]).dot(P_rect_01[:,3])
# importing left view
imgL = cv2.imread('000002_10_l.png')
# importing right view
imgR = cv2.imread('000002_10_r.png')
# importing predicted disparity view
disp = cv2.imread('000002_10_pred.png',0)
# baseline is norm of C2
b= np.linalg.norm(C1)
fx  = K_00[0,0]
fy  = K_01[1,1]
cx1 = K_00[0,2]
cx2 = K_01[0,2]
a1  = K_00[0,1] #is zero as shear is zero
a2  = K_01[0,1] #is zero as shear is zero
```

```
cy = K_00[1,2]

Q = np.eye(4, dtype='float64')
Q[0,1] = -a1/fy #is zero as shear is zero
Q[0,3] = a1*cy/fy - cx1

Q[1,1] = fx/fy
Q[1,3] = -cy*fx/fy

Q[2,2] = 0
Q[2,3] = -fx

Q[3,1] = (a2-a1)/(fy*b) #is zero as shear is zero
Q[3,2] = 1/b
Q[3,3] = ((a1-a2)*cy+(cx2-cx1)*fy)/(fy*b)
print (Q)
```

Now that we have our Q matrix we can finally use `cv2.reprojectImageTo3D` to map our 2D points into the 3D space. The following code reprojects our points and changes the color of every point to the corresponding color of our left image. It then proceeds to write the result in a `.ply` file which is used to store point cloud files.

```
points_3D = cv2.reprojectImageTo3D(disp, Q)
colors = cv2.cvtColor(imgL, cv2.COLOR_BGR2RGB)

# Delete the point if no depth is detected
mask_map = disp > disp.min()

# Mask colors and points.
output_points = points_3D[mask_map]
output_colors = colors[mask_map]
vertices = output_points
colors = output_colors
filename = 'pointcloud.ply'

colors = colors.reshape(-1,3)
vertices = np.hstack([vertices.reshape(-1,3),colors])

# write needed headers
ply_header = '''ply
        format ascii 1.0
        element vertex %(vert_num)d
        property float x
                property float y
        property float z
        property uchar red
        property uchar green
        property uchar blue
        end_header
        '''
```

```
# populate points to ply file
with open(filename, 'w') as f:
        f.write(ply_header %dict(vert_num=len(vertices)))
        np.savetxt(f, vertices, '%f %f %f %d %d %d')
```

Now that we have produced our `pointcloud.ply` successfully we need to visualize it in a 3D modeling software. Here is what a raw point cloud file looks like.

```
ply
        format ascii 1.0
        element vertex 465745
        property float x
                property float y
        property float z
        property uchar red
        property uchar green
        property uchar blue
        end_header
        -2.311160 -0.795766 2.461601 13 18 18
-2.307782 -0.795766 2.461601 13 16 17
-2.304404 -0.795766 2.461601 21 14 14
-2.301025 -0.795766 2.461601 23 16 29
-2.283286 -0.790792 2.446216 28 26 47
-2.279929 -0.790792 2.446216 48 38 46
-2.290890 -0.795766 2.461601 55 49 39
-2.331502 -0.811069 2.508939 43 49 35
-2.358293 -0.821602 2.541523 26 41 35
-2.401589 -0.837925 2.592016 27 31 25
-2.398032 -0.837925 2.592016 25 22 21
```

Figure 16: Point Cloud raw text file

As you can see from Figure 16, there are six parameters representing each point, the first three being its location in the 3D space (x,y,z) and the last three representing its color (RGB value). To visualize this `.ply` file, we will use a library called open3D. Open3D is an open-source library that is aimed at simplifying the simulation and processing of 3D data [37]. We can use the `o3d.io.read_point_cloud` function to read our point cloud followed by the `o3d.visualization.draw_geometries` function to visualize it. For visualization, Open3D includes the draw geometries() method. It accepts a list of geometries as input, produces a window, then uses OpenGL to render them all at once [37]. Here is the full code for the implementation:

```
import open3d as o3d
print("Load a ply point cloud, print it, and render it")
pcd = o3d.io.read_point_cloud("pointcloud.ply")
print(pcd)
print(np.asarray(pcd.points))
# Flip it, otherwise the point cloud will be upside down(rotate)
# and mirror right to left
pcd.transform([[1, 0, 0, 0], [0, -1, 0, 0], [0, 0, -1, 0], [0, 0, 0, 1]])
o3d.visualization.draw_geometries([pcd])
```

Running this opens a new window in our environment and renders our point cloud. Here are the results for the sample image in Figure 8.

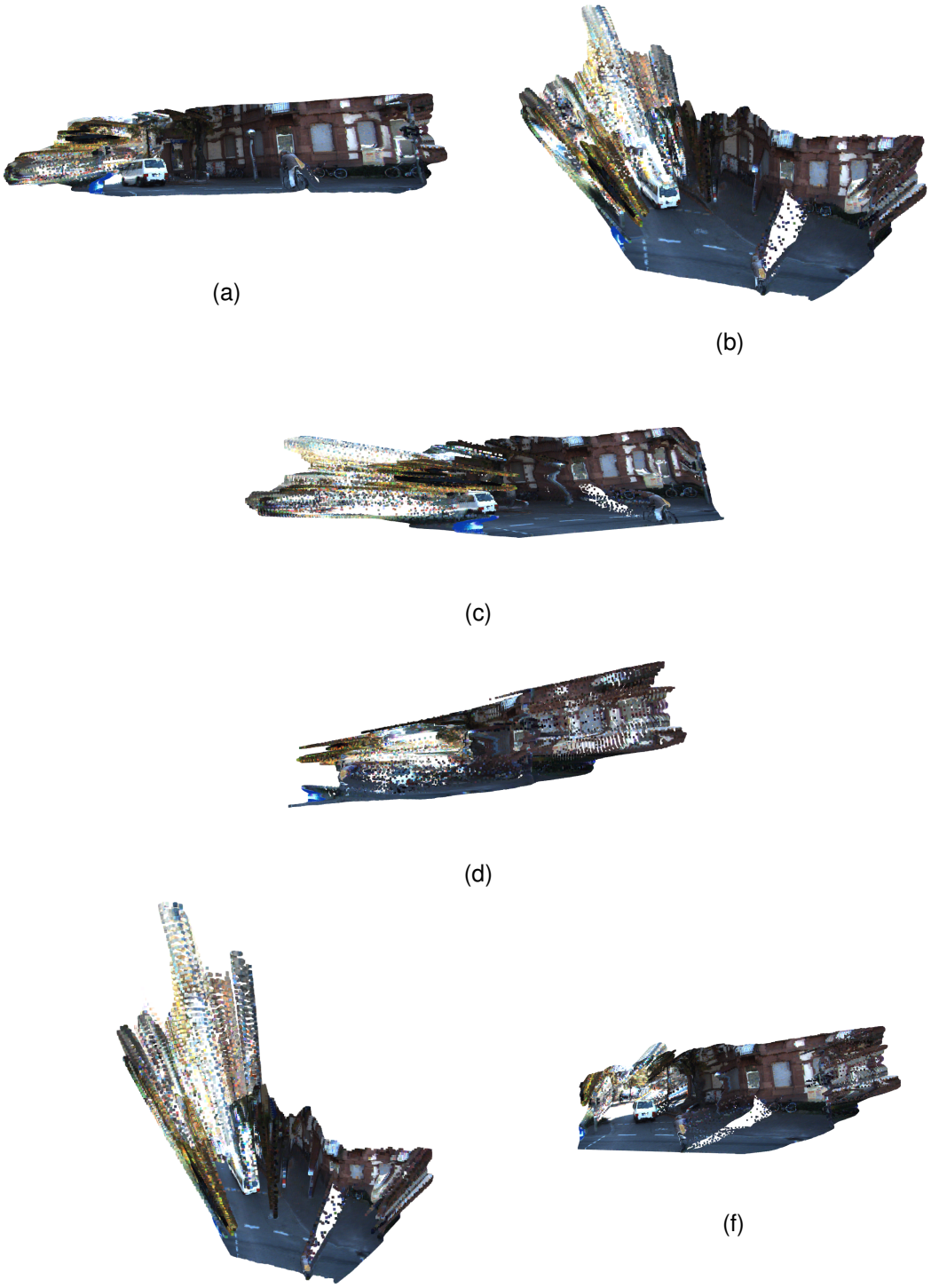


Figure 17: Point Cloud visualization for KITTI Sample Figure 8

3.4 Generating Voxel Grids

Point clouds and triangular meshes are two types of geometry that are both fluid and irregular. The voxel grid is a 3D geometry type defined on a regular 3D grid, and a voxel

can be thought of as the 3D equivalent of the pixel in 2D. Voxel grids can be worked within Open3D using the geometry type, `VoxelGrid`. The method `create_from_point_cloud` can also be used to generate a voxel grid from a point cloud. A voxel is occupied if it contains at least one point from the point cloud. The average of all the points within the voxel determines its color. The `voxel_size` option specifies the resolution of the voxel grid. [37][32]. A voxel can be described as a monocolored block of pixels.

```
# pcd is our point cloud
# 0.05 is voxel size
voxel_grid = o3d.geometry.VoxelGrid.create_from_point_cloud(pcd,
voxel_size=0.05)
o3d.visualization.draw_geometries([voxel_grid])
```

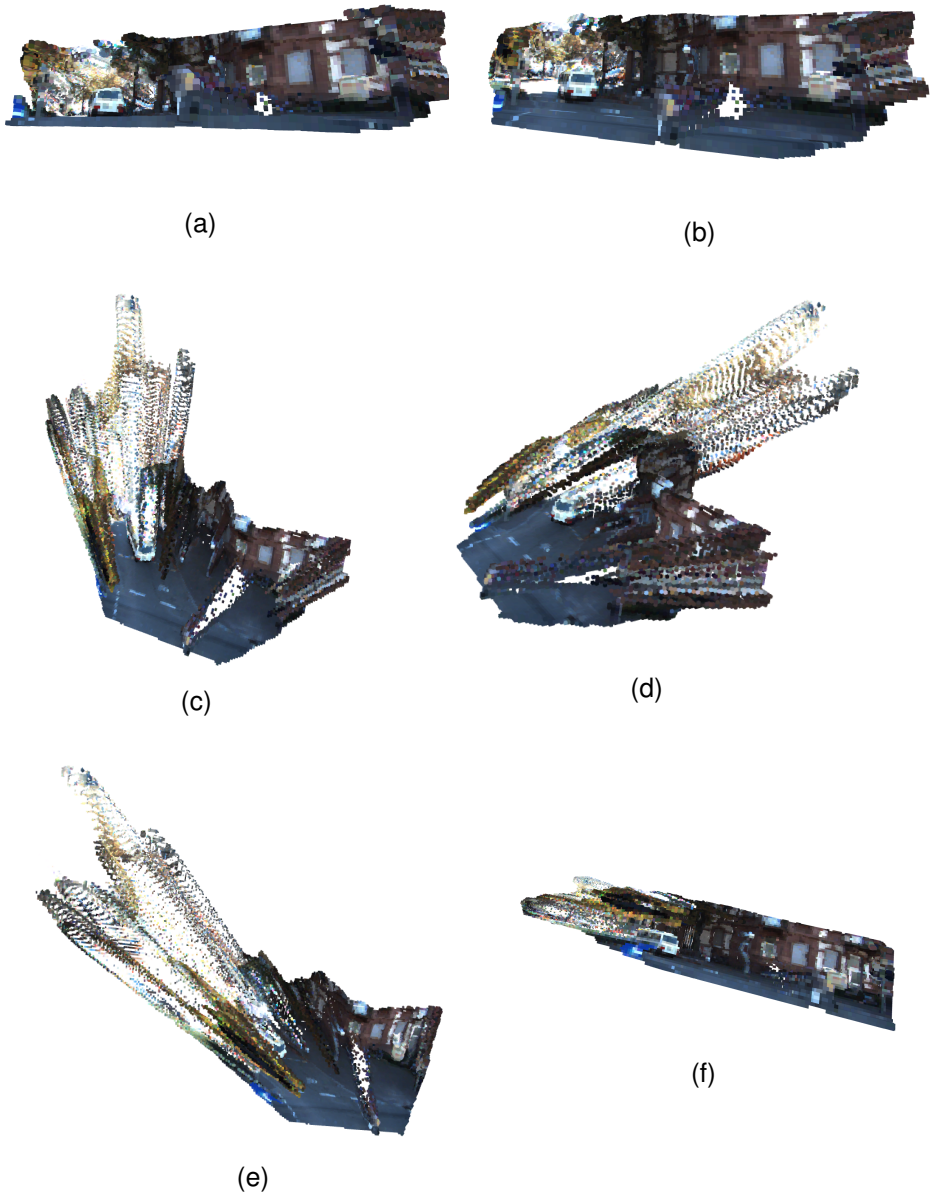


Figure 18: Voxel Grid = 0.05 visualization for KITTI Sample Figure 8

It is noticeable from the figures above that the visualization is made from monocolored blocks instead of pixels. The size of these blocks can be manipulated with the `voxel_size` variable. Down below is shown a comparison of a portion of a point cloud, and voxel grids with different voxel sizes.

(a) point cloud

(b) voxel_size = 0.025

(c) voxel_size = 0.05

(d) voxel_size = 0.075

(e) voxel_size = 0.1

(f) voxel_size = 0.25

Figure 19: A point cloud and voxel grids with different voxel sizes

4 Evaluation of the Investigation

In this section, we will be discussing the results, performance, and methodology used throughout this research. We first mention and explain some properties of our results and then go on to describe some of the challenges encountered.

4.1 Run-Time Analysis

MobileStereoNet comes with a `cost.py` file which lets us compute the complexity of our network in terms of the number of operations and parameters. The number of operations is always in whole numbers. The second property is the number of free parameters in a neural network structure, such as the number of neurons and the number and strength of connections between neurons (weights). Both these properties are used to determine the network's complexity [26].Complexities for both the 2D and 3D models are provided.

```
==========================
 2D-MobileStereoNet
==========================
Number of operations:           32.0
Number of parameters:           2.228
==========================
 3D-MobileStereoNet
==========================
Number of operations:           152.0
Number of parameters:           1.77
```

Figure 20: Complexities in terms of operations and parameters

During this research, we used Google Collab Pro with NVIDIA Tesla P100 GPU 16GB memory. Training our model on 160 image sets and validating them on 40 image sets took about 8 hours for 400 epochs on this system. An image set consists of the left view, right view, and disparity ground truth.

```
/content# nvidia-smi
Tue May  3 18:57:47 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   36C    P0    26W / 250W |      0MiB / 16280MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
/content#
```

Figure 21: GPU Info by nvidia-smi

4.2 Point Cloud Density

Point cloud density refers to the number of coordinates collected per unit area. When point cloud data is processed and turned into a 3D digital model, these coordinates act like pixels. An excessively high point cloud density might be unnecessary in our case since what we are trying to achieve is a 360 wide view for an autonomous vehicle. Taking this into account, the point cloud density we have achieved seems to be sufficient. Lower density signifies less information (low resolution), while higher density means more information (high resolution). It is crucial to understand point cloud density because it can affect the quality of data collected. For the point cloud we have generated at Figure 17, we have got 465745 points. This can be seen from the number of lines from our raw text ply file as every line represents one point. Point cloud density can be statistically described in three main ways. They are: [7]

the number of neighbors N points counted inside a sphere of radius R),
the surface density: number of neighbors divided by the neighborhood surface = $N/(Pi.R2)$,
the volume density: number of neighbors divided by the neighborhood volume = $N/(4/3.Pi.R3)$

For calculating and visualizing these properties we will be using a software called CloudCompare. It is a 3D point cloud software used to perform analysis on point clouds. By performing point cloud density estimation and finding the Gaussian mean on our sample point cloud, we get the following results. The next figure shows the point cloud density map with colors corresponding to the histograms in Figure 23.

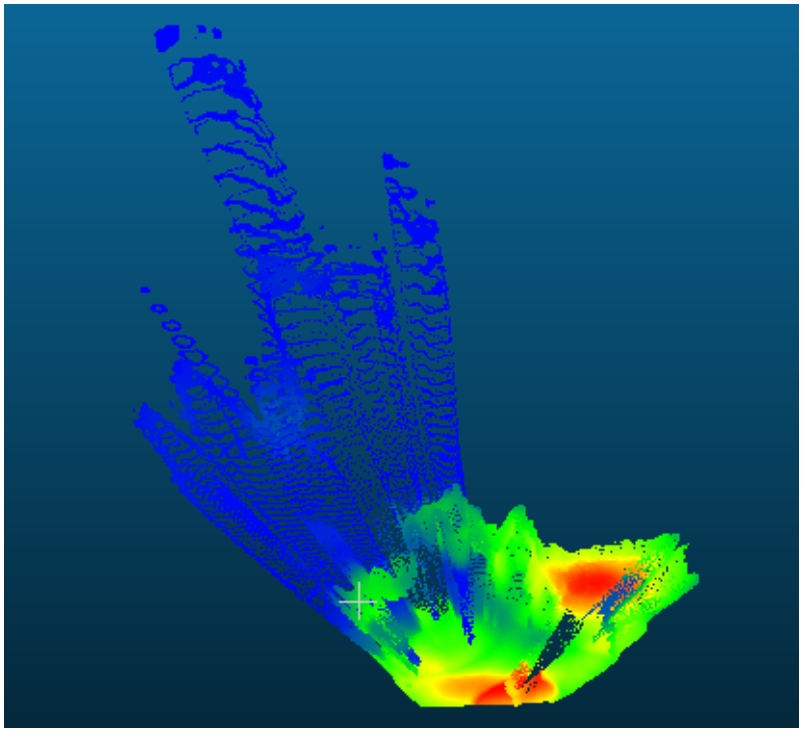
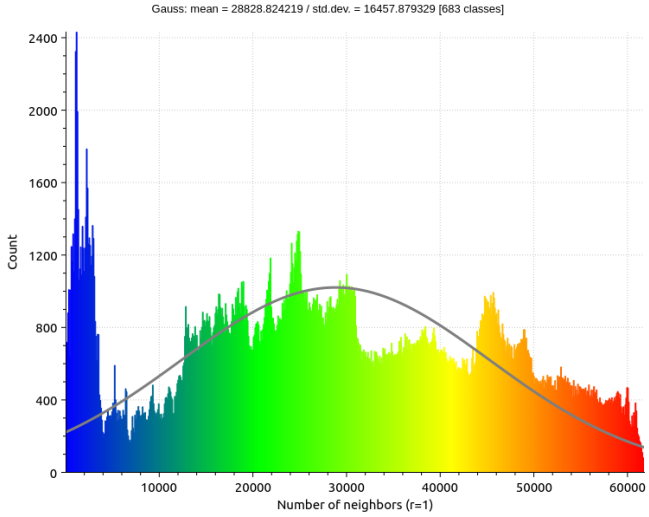
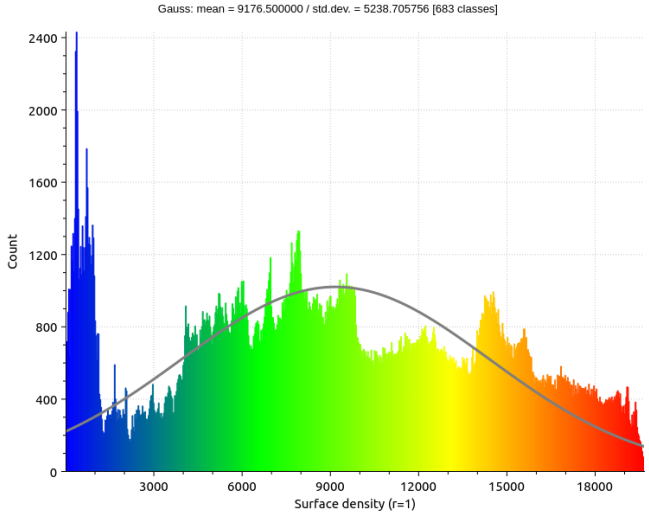


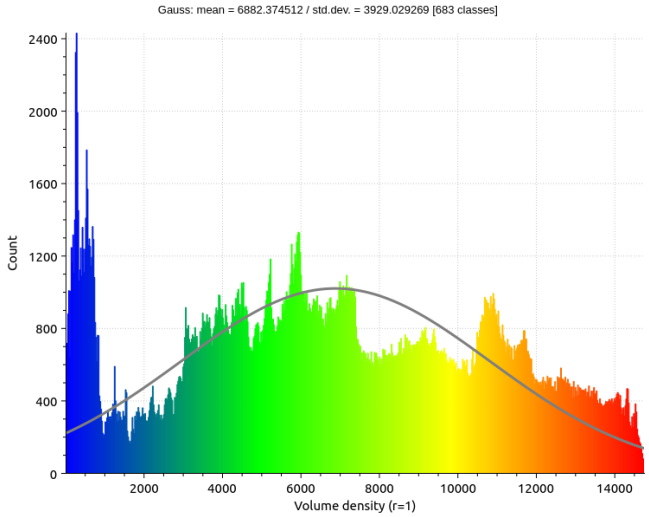
Figure 22: Point cloud density map



(a) Histogram nearest neighbours, Gaussian mean = 28828.824219



(b) Histogram surface density, Gaussian mean = 9176.5



(c) Histogram volume density, Gaussian mean = 6882.374512

Figure 23: Point cloud Density analysis

The histograms have the same shape as they are all based on the number of nearest neighbors property within a sphere of radius R. What differs between them is the context they are put in. For 23a, simply the number of nearest neighbors is taken while in 23b and 23c the number of nearest neighbors is scaled by the standard formulas for a sphere's surface area and volume respectively. From the density map, we can generalize that closer objects have a higher density while the density declines the farther the points are.

4.3 Qualitative and Quantitative Assessment

The KITTI benchmark has an evaluation server which computes the percentage of bad pixels averaged over all ground truth pixels for all test images. A pixel is considered to be correctly estimated if the disparity or flow end-point error is $<$3px or $<$5 [30].

Rank	Method	D1-bg	D1-fg	D1-all	Density	Runtime	Environment
97	3D-MSNet / MSNet3D	1.75%	3.87%	2.10%	100.00%	1.5s	Python,1080Ti
166	2D-MSNet / MSNet2D	2.49%	4.53%	2.83%	100.00%	0.4s	Python,1080Ti

D1-bg is percentage of stereo disparity outliers in first frame averaged only over background regions,
D1-fg is percentage of stereo disparity outliers in first frame averaged only over foreground regions, and
D1-all is percentage of stereo disparity outliers in first frame averaged over all ground truth pixels. Here is another table comparing Mobilesteronet with other methods.

Method	EPE(px)	D1(%)	px-3(%)	MACs(G)	Params(M)
PSMNet[2]	0.88	2.00	2.10	256.66	5.22
GA-Net-deep[32]	0.63	1.61	1.67	670.25	6.58
GA-Net-11[32]	0.67	1.92	2.01	383.42	4.48
GwcNet-gc[6]	0.63	1.55	1.60	260.49	6.82
GwcNet-g[6]	**0.62**	**1.49**	**1.53**	246.27	6.43
2D-MobileStereoNet	0.79	2.53	2.67	**32.2**	2.32
3D-MobileStereoNet	0.66	1.59	1.69	153.14	**1.77**

Table 5: Comparison on KITTI 2015 validation set.

Methods	All(%)			Noc(%)		
	$D1_{bg}$	$D1_{fg}$	$D1_{all}$	$D1_{bg}$	$D1_{fg}$	$D1_{all}$
MC-CNN[31]	2.89	8.88	3.89	2.48	7.64	3.33
Fast DS-CS[29]	2.83	4.31	3.08	2.53	3.74	2.73
GCNet[10]	2.21	6.16	2.87	2.02	5.58	2.61
DeepPruner-Fast[4]	2.32	3.91	2.59	2.13	3.43	2.35
PSMNet[2]	1.86	4.62	2.32	1.71	4.31	2.14
AutoDispNet-CSS[22]	1.94	**3.37**	2.18	1.80	**2.98**	2.00
DeepPruner-Best[4]	1.87	3.56	2.15	1.71	3.18	1.95
GwcNet-g[6]	**1.74**	3.93	2.11	**1.61**	3.49	**1.92**
2D-MobileStereoNet	2.49	4.53	2.83	2.29	3.81	2.54
3D-MobileStereoNet	1.75	3.87	**2.10**	**1.61**	3.50	**1.92**

Figure 24: Comparison on KITTI 2015 benchmark. 3DMobileStereoNet requires 98% and 72% fewer parameters compared to AutoDispNet-CSS and GwcNet-g, respectively [27]

This figure shows a sample image from the KITTI dataset followed by predictions and error maps by the 2D and 3D MobileStereoNet model.

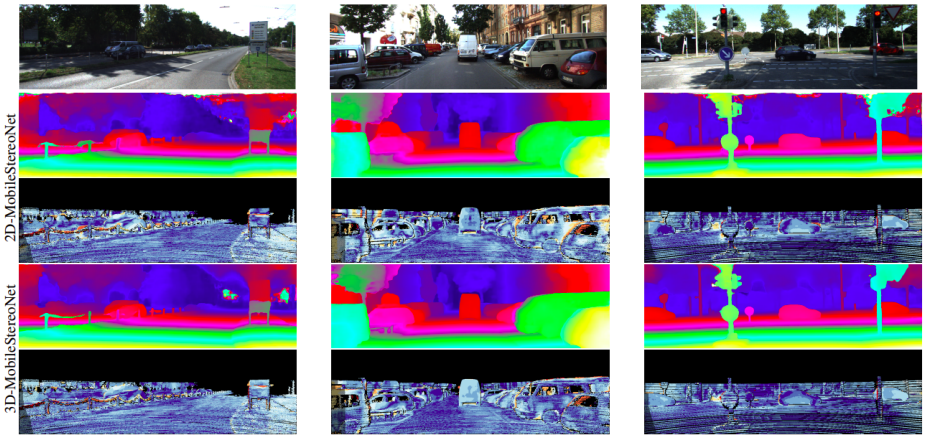


Figure 25: Sample Image set from KITTI, predictions and error maps on MSNet2D and MSNet3D [27]

5 Conclusions

This paper has presented a pipeline for the stereo processing of driving images from the first step of disparity estimation to the visualization of the computed 3D models. We started by outlining alternative methods for disparity and depth estimation along with their setbacks which set the ground for proposing a new method for disparity estimation using MobileStereoNet. We have proposed this method mainly because of its lightweight and computationally inexpensive nature. This stems from its use of MobileNets, a neural network architecture based on depthwise separable convolutions.

We started with fine-tuning our pre-trained model to make it more robust and specific for our dataset. We then predicted the disparity map for our pair of images using our fine-tuned model. Leveraging the KITTI calibration file, we found out the intrinsic and extrinsic parameters of the cameras used for data collection. These parameters are important for us to calculate the transformation matrix needed to convert our disparity data into depth data and then reproject it into the 3D space.

We used the framework Open3D to visualize our 3D models. For visual presentation of our models, we explored the options of point clouds and voxel grids. We explained the fundamental intersection and union between these two methods and also tried to customize them to our needs by optimizing different input parameters. We also experimented with different transformation matrix computation techniques.

We then analyzed our model by computing the complexities of our model in terms of operations and parameters followed by evaluating the point cloud density map. The histograms for the density values in relationship with the surface area and volume of a sphere were provided. The Gaussian mean and standard deviation for each histogram has been calculated to give a single unit insight into the point cloud density of our model.

Finally, we compared MobileStereoNets to various popular models, finding that they outperformed them in terms of size, speed, and accuracy. We also had a look at the KITTI benchmark to see how MobileStereoNet compares to other lightweight and non-lightweight models with error and time complexity. An error map computed from the predicted disparity values and ground truth data has also been provided.

In this study, we have tried to ease the classic, age-long challenge of depth estimation. We have leveraged recent developments in lightweight neural networks to estimate the depth and then produce a 3D model of driving scenes with reasonable speed, accuracy, and time complexity. This very well supports the process of reducing the expensive, non-robust stereo rig found in autonomous driving systems.

References

[1] Naveen Appiah and Nitin Bandaru. “Obstacle detection using stereo vision for self-driving cars”. In: *IEEE Intelligent Vehicles Symposium (IV)*. 2011, pp. 926–932.

[2] Martin Asenov and Dimitar Dimitrov. “Generating disparity maps using Convolutional Neural Networks”. In: ().

[3] *Autonmous cars benefits*. https://www.synopsys.com/automotive/what-is-autonomous-car.html. Accessed: 2010-09-30.

[4] Walter Brenner and Andreas Herrmann. “An overview of technology, benefits and impact of automated and autonomous driving on the automotive industry”. In: *Digital marketplaces unleashed* (2018), pp. 427–442.

[5] Jason Brownlee. “What is the Difference Between a Batch and an Epoch in a Neural Network”. In: *Machine Learning Mastery* 20 (2018).

[6] *Camera Calibration*. https://www.mathworks.com/help/vision/ug/camera-calibration.html. Accessed: 2010-09-30.

[7] *Cloud Compare Documentation*. https://www.cloudcompare.org/doc. Accessed: 2010-09-30.

[8] Pinliang Dong and Qi Chen. *LiDAR remote sensing and applications*. CRC Press, 2017.

[9] Andreas Geiger et al. “Vision meets robotics: The kitti dataset”. In: *The International Journal of Robotics Research* 32.11 (2013), pp. 1231–1237.

[10] *groupcalib*. https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html. Accessed: 2010-09-30.

[11] Yulan Guo et al. “Deep Learning for 3D Point Clouds: A Survey”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43.12 (2021), pp. 4338–4364. DOI: 10.1109/TPAMI.2020.3005434.

[12] Rostam Affendi Hamzah and Haidi Ibrahim. “Literature survey on stereo vision disparity map algorithms”. In: *Journal of Sensors* 2016 (2016).

[13] Andrew G Howard et al. “Mobilenets: Efficient convolutional neural networks for mobile vision applications”. In: *arXiv preprint arXiv:1704.04861* (2017).

[14] Reinhard Koch and Johannes Bruenger. “Depth estimation”. In: *Computer Vision: A Reference Guide* (2020), pp. 1–5.

[15] Deepika Kumari and Kamal Kaur. “A Survey on Stereo Matching Techniques for 3D Vision in Image Processing”. In: *International Journal of Engineering and Manufacturing* 6 (July 2016), pp. 40–49. DOI: 10.5815/ijem.2016.04.05.

[16] Deepika Kumari and Kamaljit Kaur. “A survey on stereo matching techniques for 3D vision in image processing”. In: *Int. J. Eng. Manuf* 4 (2016), pp. 40–49.

[17] Nalpantidis Lazaros, Georgios Christou Sirakoulis, and Antonios Gasteratos. “Review of stereo vision algorithms: from software to hardware”. In: *International Journal of Optomechatronics* 2.4 (2008), pp. 435–462.

[18] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), pp. 436–444.

[19] Chi Li and Zhiguo Cao. “Lidar-stereo: Dense depth estimation from sparse lidar and stereo images”. In: *Proceedings of the 2020 5th International Conference on Multimedia Systems and Signal Processing*. 2020, pp. 11–15.

[20] CH Lo and Alan Chalmers. “Stereo vision for computer graphics: the effect that stereo vision has on human judgments of visual realism”. In: *Proceedings of the 19th spring conference on Computer graphics*. 2003, pp. 109–117.

[21] Markus Maurer et al. *Autonomous driving: technical, legal and social aspects*. Springer Nature, 2016.

[22] Gerard Medioni and Ramakant Nevatia. “Segment-based stereo matching”. In: *Computer Vision, Graphics, and Image Processing* 31.1 (1985), pp. 2–18. ISSN: 0734-189X. DOI: https://doi.org/10.1016/S0734-189X(85)80073-6. URL: https://www.sciencedirect.com/science/article/pii/S0734189X85800736.

[23] Prathik Naidu Mihir Garimella. “Beyond the pixel plane: sensing and learning in 3D”. In: *The Gradient* (2018).

[24] Lazaros Nalpantidis and Antonios Gasteratos. “Stereovision-based fuzzy obstacle avoidance method”. In: *International Journal of Humanoid Robotics* 8.01 (2011), pp. 169–183.

[25] Whitman Richards and Martin G. Kaye. “Local versus global stereopsis: Two mechanisms?” In: *Vision Research* 14.12 (1974), pp. 1345–1347. ISSN: 0042-6989. DOI: https://doi.org/10.1016/0042-6989(74)90008-X. URL: https://www.sciencedirect.com/science/article/pii/004269897490008X.

[26] Johann Schumann, Pramod Gupta, and Yan Liu. “Application of neural networks in high assurance systems: A survey”. In: *Applications of neural networks in high assurance systems*. Springer, 2010, pp. 1–19.

[27] Faranak Shamsafar et al. “MobileStereoNet: Towards Lightweight Deep Networks for Stereo Matching”. In: *2022 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*. 2022, pp. 677–686. DOI: 10.1109/WACV51458.2022.00075.

[28] Nima Tajbakhsh et al. “Convolutional neural networks for medical image analysis: Full training or fine tuning?” In: *IEEE transactions on medical imaging* 35.5 (2016), pp. 1299–1312.

[29] Tangfei Tao, Ja Choon Koo, and Hyouk Ryeol Choi. “A fast block matching algorithm for stereo correspondence”. In: *2008 IEEE Conference on Cybernetics and Intelligent Systems*. 2008, pp. 38–41. DOI: 10.1109/ICCIS.2008.4670774.

[30] *The KITTI Vision Benchmark Suite*. http://www.cvlibs.net/datasets/kitti/eval_scene_flow.php?benchmark=stereo.

[31] Edna Chebet Too et al. “A comparative study of fine-tuning deep learning models for plant disease identification”. In: *Computers and Electronics in Agriculture* 161 (2019), pp. 272–279.

[32] *Voxelization Open 3D*. http://www.open3d.org/docs/latest/tutorial/Advanced/voxelization.html. Accessed: 2010-09-30.

[33] Yan Wang et al. “Pseudo-LiDAR From Visual Depth Estimation: Bridging the Gap in 3D Object Detection for Autonomous Driving”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2019.

[34] Yihong Wu, Fulin Tang, and Heping Li. “Image-based camera localization: an overview”. In: *Visual Computing for Industry, Biomedicine, and Art* 1.1 (2018), pp. 1–13.

[35] Qingxiong Yang et al. “Real-time Global Stereo Matching Using Hierarchical Belief Propagation”. In: vol. 6. Jan. 2006, pp. 989–998. DOI: 10.5244/C.20.101.

[36] Guangjun Zhang, Junji He, and Xiuzhi Li. “3D vision inspection for internal surface based on circle structured light”. In: *Sensors and Actuators A: Physical* 122.1 (2005). SSSAMW 04, pp. 68–75. ISSN: 0924-4247. DOI: https://doi.org/10.1016/j.sna.2005.04.012. URL: https://www.sciencedirect.com/science/article/pii/S0924424705002347.

[37] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. “Open3D: A Modern Library for 3D Data Processing”. In: *arXiv:1801.09847* (2018).