ECE 188 SP23 – Project Writeup
Haolin Xiong

Stage 1: Image Denoising

In the first part of the project, several signal processing image denoising techniques are investigated and tested on five types of noise. One best filter is decided for each kind of noisy image based on preliminary testing of 1 image. Then, with extensive grid search hyperparameter tunning, the average PSNR, SSIM, as well as the best parameter settings are shown in the table 1 below. Examples of images before and after denoising are shown as Figure 1 and 2. All the filters are implemented using the *opencv* library.

Table 1. Benchmark Result for Denoised Images

Noise Type	Best Filter	Best Parameter	Average PSNR	Average SSIM
Gaussian Noise	Gaussian Blur	Kernal Dim: (21,11)	20.48	0.50
Gaussian Blur	Bilateral Filter	Sigma (Color/Space): 50, d: -1	19.18	0.47
Salt & Pepper	Median Filter	Kernal Size: 3	26.77	0.89
Speckle	Non-Local Means Filter	Block Size: 15	22.76	0.72
Motion Blur	Wiener Deconvolution	Kernal Size: 15, Angle Specific to Each Image	19.88	0.63

(Note: Filters are applied to each color channel separately)

From the result, we can see one of the biggest limitations of traditional signal processing filter is that different type of noise requires different type of filter. We also observe that the same parameter settings perform drastically across different images with the same noise type. It is hard to find a single best filter and setting to efficiently deal with various kinds of noises lay in real world problems with only mathematics or signal processing techniques.

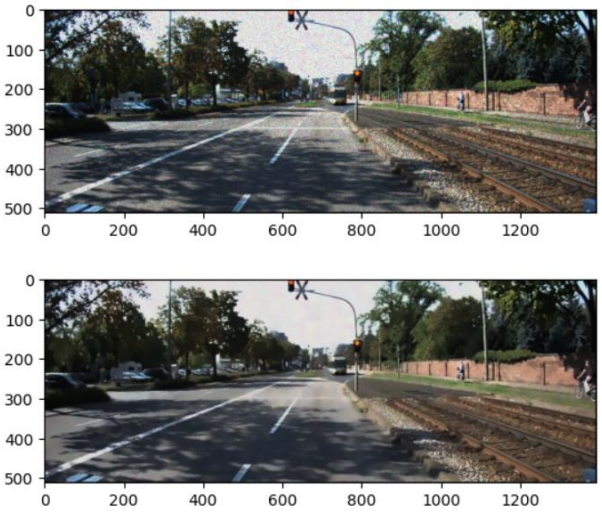


Figure 1. Image with Salt & Pepper Noise Before and After Denoising

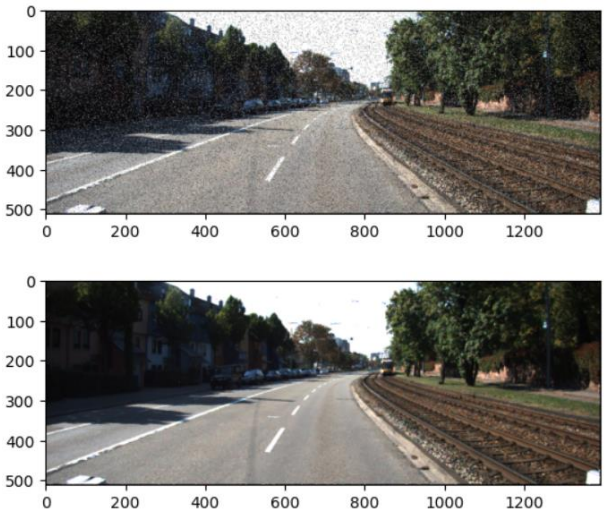


Figure 2. Image with Speckle Noise Before and After Denoising

Stage 2: Rectification & Stereo Matching

The second part of the project is broken down into two sections: Rectification task and Stereo Matching task. Similar to the previous stage, we aim to solve the problems without using Deep Learning technique.

(A). Rectification

Rectification plays an important role in computer vision and image analysis such as Stereo Matching and Image Stitching. It aligns different cameras' perspective to provide a uniform geometric representation of a scene. In this task, we are given 11 warped images and are expected to find corresponding homography to warp them back to their original state.

There are some common steps to follow to rectify an image. First, we need to use an image descriptor (SIFT) to find matching pixels or features in the original and the warp image. Then, we apply *RANSAC* to the matching features to estimate the best homography for inverse warping. Finally, we warp the image and benchmark the pipeline with PSNR/SSIM.

With this process, the average PSNR achieved across 11 images is 31.5 and the average SSIM is 0.91. Figure 3 & 4 below show two example results.

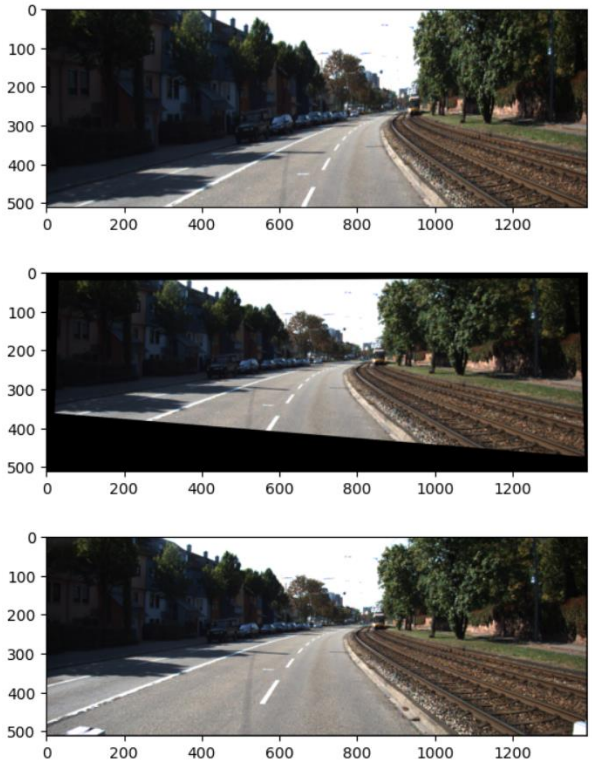


Figure 3. Warped, Rectified, and Original Image Example 1

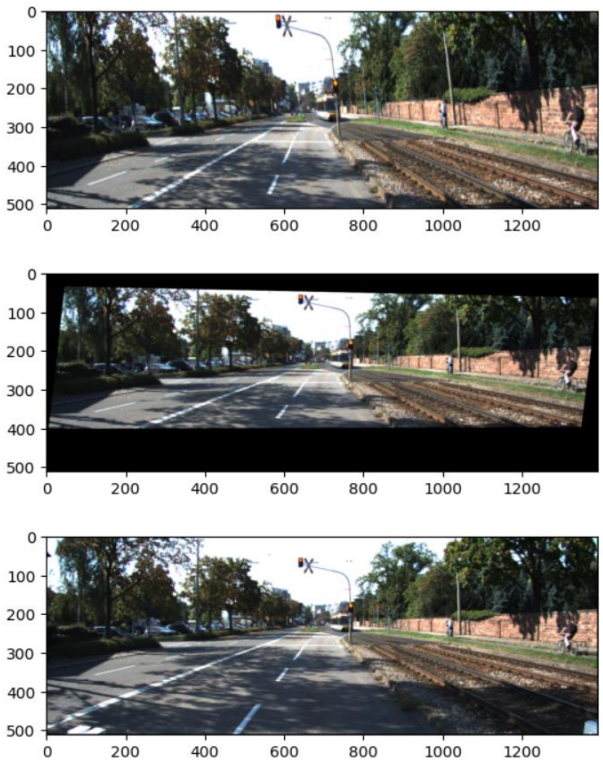


Figure 4. Warped, Rectified, and Original Image Example 2

(B). Stereo Matching

Stereo matching is the process of extracting object's depth information using one image taken from two viewpoints. It allows 3D reconstruction of a scene from 2D images. In this task, I explore the two classical stereo matching algorithms: Stereo Block Matching (BM) and Semi-Global Block Matching (SGBM). For both methods, I choose the *opencv* Python library for straight forward implementation and parameter testing.

Block matching is done by cutting the image into small squares then search the squares along the same horizontal line on the other image to find potential matches. A match is found by minimizing the RMSE between corresponding blocks in the two images and the disparity value is calculated as the horizontal pixel shift. Block matching is simple and fast but sometimes not reliable because the disparity of all the pixels in the same block is assumed to be the same, which is not the case in real world images especially around edges and corners. For this task, with extensive grid search on block size, the lowest pixel RMSE achieved is 2.47 with a density of 0.0395. Figure 5 below shows an example output.

One the other hand, Semi-Global Block Matching is a more sophisticated algorithm. It considers both the local block matching and some image-wise global information. First, it searches for matching blocks the same way as a regular BM. Then, it takes in account for the cost on both the current blocks and their neighboring blocks by trying to minimize the sum of these costs while adding penalties for large changes in disparity. The process is then repeated in several directions in contrast of that in the regular BM. The essential idea behind SGBM is to introduce smoothness in the disparity map by taking neighboring information into account. It is far less computationally intensive than true global method but still requires more computation than a regular BM. For our task, also with extensive grid search, the lowest RMSE achieved for the same image used in BM is 3.51 with density 0.131. Figure 6 below shows the example output.

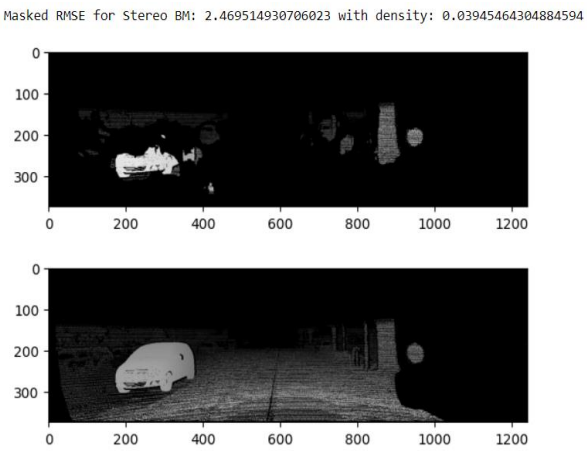


Figure 5. Disparity from BM and the Ground Truth

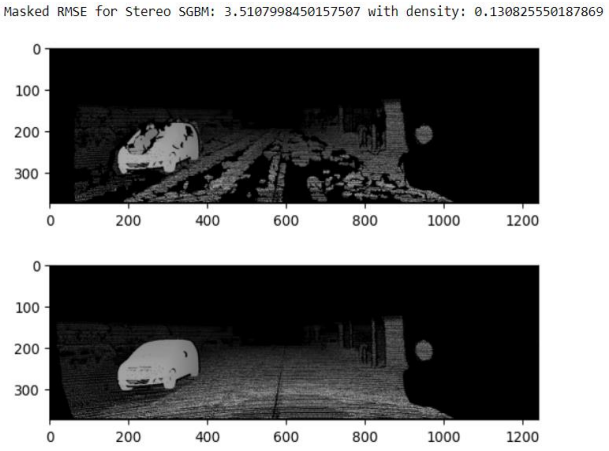


Figure 6. Disparity from SGBM and the Ground Truth

Even though the RMSE found from SGBM is higher than that from BM, from the actual images we can still state that SGBM is better. The higher RMSE is due to the masked calculation method we choose. When taking density into account, the total error of SGBM is actually lower than that of BM. (i.e., 3.51/0.131 < 2.47/0.0395) Figure 7 shows the disparity map for all 5 images generated by SGBM.

Figure 7. Disparity from SGBM and the Ground Truth by Image

Stage 3: Point Cloud Data Object Detection

In this part, our goal is to implement a state-of-the-art objection detection architecture and train it with the Point Cloud dataset from Kitti to perform 3D objection detection (i.e., Draw 3D bounding box) task. At first, I tried to use **PointNet** [Qi et al. in 2017] but I ran into numerous troubles while downloading dependencies especially TensorFlow which had a huge v1 to v2 migration. Then, I came across **PointPillars** [Lang et al. in 2018], a very novel, lightweight, and PyTorch-based 3D object detection architecture which also achieves exceptional results. In this stage, I will be implementing the PointPillars architecture focusing on drawing 3D bounding boxes for “Pedestrian”, “Cyclist”, and “Car”.

Literature Review:

PointNet:

PointNet, introduced by Qi et al. in 2017, proposed on novel way of directly processing point clouds data. Before PointNet, most researches were based on volumetric approaches or 2D hyperplane projection for object detection and similar tasks. But these approaches typically involved large computations or would forfeit too much 3D information.

On the other hand, PointNet operates directly on point clouds, making it possible to process unordered 3D points while still preserving detail local structure and global context. It achieves this by applying T-Net, which is consisted of several multi-layer perceptron (MLP), to each point individually as feature extractors and then aggregating the results with max pooling. The T-Nets extract and align features from each point individually and the symmetric max pooling aggregate the extracted features. The two combined allows PointNet to be permutation invariant while still reserves some global information.

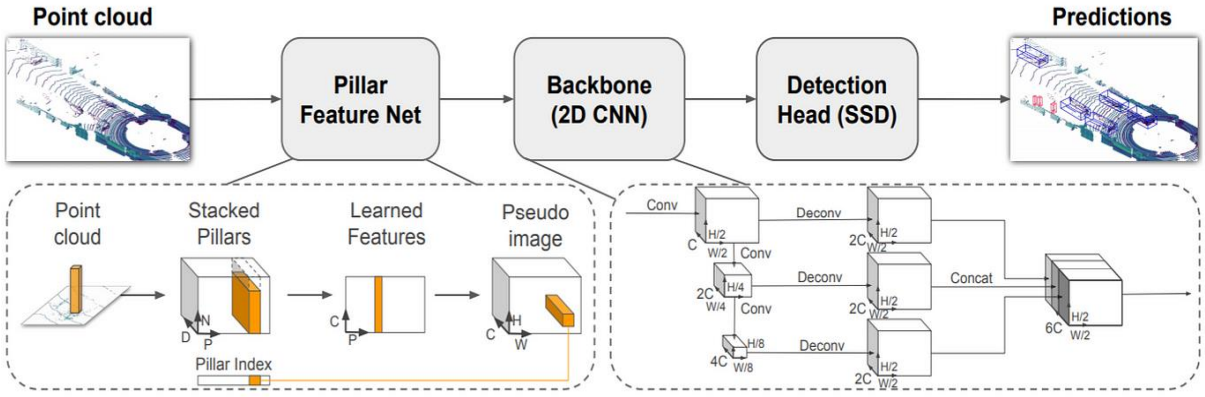
PointPillars:

PointPillars, introduced by Lang et al. in 2018, proposed an efficient encoding framework of point cloud data for visual related tasks. While PointNet operates directly on the raw point clouds, PointPillars takes a slightly different approach by discretizing the point cloud into a pseudo-image (pillars), which greatly reduces the complexity of the data.

Specifically, PointPillars divides the 3D point cloud space into vertical columns (pillars) and encodes the points within a pillar into a feature vector. The 2D feature pillars then serves as input to a standard 2D backbone convolutional network for bounding box detection. Its simplicity and high efficiency make it suitable for real-time applications, such as autonomous driving. According to the author, the standard PointPillars algorithm can easily achieve a 60Hz object box detection and the faster version can even achieve as high as 105Hz.

The Architecture:

In this project, I aimed to focus on training a PointPillars model for 3D bounding box drawing problem. The PointPillars architectures can be broken down into **4** steps. Figure 8 shows an overview flowchart of the architecture.



Picture Ref link: https://becominghuman.ai/pointpillars-3d-point-clouds-bounding-box-detection-and-tracking-pointnet-pointnet-lasernet-67e26116de5a

Figure 8. PointPillars Architecture

(1). Preprocessing and Pillar Encoding

The PointPillars leverages the efficiency of 2D CNN backbone so the first step in is to preprocess the point cloud data and transform it into a suitable format.

The 3D space is divided into equal-sized vertical columns or "pillars". Each pillar holds a subset of the input points. The pillars have the same width/length but their height spans the full range of z in the point cloud. For each point in a pillar, a four-element vector is created: x, y, and z coordinates (normalized relatively to the pillar's center), and its reflectance value. These vectors are then fed into a shared MLP to produce point-wise features. Next, the pillar-wise features are computed by applying a max pooling layer to each pillar, introducing permutation invariance.

(2). Spatial Feature Encoding

Once the pillar features are obtained, they are organized into a pseudo-image. This is a 2D grid, where each cell corresponds to a pillar and contains the pillar's feature vector. Empty pillars are filled with zeros. This grid forms a representation similar to a bird-eye view of the input data.

(3). Backbone Network

The pseudo-images are then passed through a backbone 2D convolutional network to extract higher-level features. This network could be any standard 2D object detection architecture such as ResNet or VGG. In this project, I just used several Conv2D-ReLu-BatchNorm layers for simplicity. This stage is where PointPillars leverages the efficiency of highly optimized 2D convolution networks.

(4). Object Detection & Bounding Box Drawing

The output from the backbone network is fed into a detection head, which predicts the final bounding boxes and class labels. The detection head consists of several convolutional layers, followed by two parallel fully-connected subnetworks. One subnetwork predicts bounding box parameters (size, location, and orientation), while the other subnetwork predicts the class scores for each box.

This is done by sliding a small window across the feature map from the backbone network and making predictions for each window position. The predictions are then decoded into 3D bounding boxes in the original point cloud space. This process includes some applications of anchor boxes which I will not get into too much details about.

In summary, PointPillars preserves much of the 3D information while leveraging the strengths of 2D convolutional neural networks. However, like other projection-based methods, it can lose some 3D details due to the encoding process.

The Implementation & Hyperparameters:

The loss function contains three parts, the bounding box overlapping loss, the bounding box classification loss, and the bounding box direction loss where each defines an error aspect of the bounding box drawing problem.

The original training data are split 50-50 into training and validation set. Then, data augmentation techniques such as mirroring and rotation are applied to up sample the training set back to around 7500 images. With a batch size of 32, each epoch should consist of around 230 iterations.

The model training hyperparameters is selected to give the best 3D bounding box accuracy by running 10 epochs. The learning rate is selected to be 0.001, which allowed the model to learn at a steady pace without overshooting or oscillating too much. In preliminary testing, this learning rate is sufficient to guide the model towards a smooth convergence.

With these hyperparameters setting, with a RTX 4090 graphics card and an Intel-i9 processor, each epoch takes around 2 minutes. The whole 200-epoch training processing takes around 7 hours. The figure below shows that epoch 199 has 232 iterations which matches our calculation earlier.

Figure 9. Training Example

Model Evaluation & Output Examples

The three main target classes I focus here are Car, Pedestrian, and Cyclist. The task for each class also comes with easy, medium and hard difficulty setting where each has a different correctness constraint. For example, the bounding box drawn has to overlap with the ground-truth more for the box to be counted as correct in hard mode then that in easy mode. The validation accuracy by epoch plots for identifying different classes in easy mode is shown as Figure 10 below for demonstration purposes.

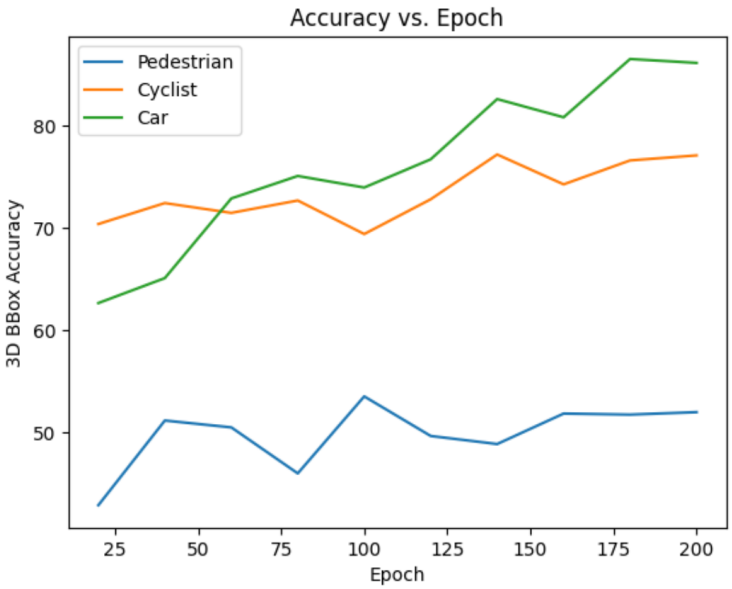


Figure 10. 3D Bound Box Drawing Accuracy vs. Epoch

From the plot above, we can see that the pedestrian class is definitely more difficult to be identified than the other two. A plateau in the accuracy for pedestrian bounding happens at around 160 Epoch while the accuracy for the other two classes is still increasing. If we were to let the model run for longer or try out more hyper-parameters, the accuracy should increase further. Figure 11 below shows the final validation accuracy for each class at each difficulty level (Easy-Medium-Hard from left to right).

```
==========BBOX_3D==========
Pedestrian AP@0.5: 51.9968 46.6846 43.3879
Cyclist AP@0.5: 77.1086 60.3285 56.3210
Car AP@0.7: 86.1647 76.8893 73.8190
```

Figure 11. Model Validation at Epoch 200

Figure 12 below shows the bird-eye pillar view perceived by our classifier and Figure 13 shows the corresponding image and 3D bounding boxes. The yellow boxes are the ground truth values provided by Kitti and the blue boxes are drawn by our model. We can see that even though the large red van and the black van on the left are not labeled in the ground truth set, our model can still pick them up and correctly classify them as cars. This shows the generalizability and the advantage of leveraging 2D convolution in PointPillars model. Figure 14 shows another example output from passing in image #134 through the model (Red: Pedestrian, Green: Cyclist, Blue: Car).

In short summary, the stage 3 of the project has been fun and educational. I learn that there are many clever perspectives for viewing the same problem which can leverage the strengths of both SOTA architectures and of older but more reliable ones. In the context of point cloud 3D bounding box problem, there is still a long journey to be explored before the model achieving super-human level in all classes.

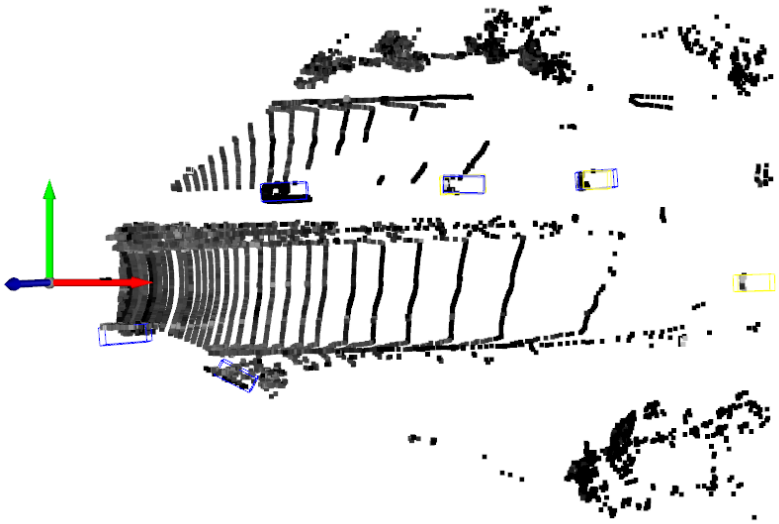


Figure 12. Pseudo-Image of #297

Figure 13. 2D image and 3D Bounding Box from #297

Figure 14. 2D image and 3D Bounding Box from #134

Reference:

The Kitti Vision Benchmark Suite. (n.d.).
https://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=3d

Lang, A. H. et al. Pointpillars: Fast encoders for object detection from point clouds.
https://openaccess.thecvf.com/content_CVPR_2019/papers/Lang_PointPillars_Fast_Encoders_for_Object_Detection_From_Point_Clouds_CVPR_2019_paper.pdf

DataScienceUB. (2022, April 25). *PointNet implementation explained visually*. Medium.
https://datascienceub.medium.com/pointnet-implementation-explained-visually-c7e300139698

Tyagi, A. (2020, September 28). *Pointpillars - 3D point clouds bounding box detection and tracking (PointNet, pointnet++, LaserNet...* Medium. https://becominghuman.ai/pointpillars-3d-point-clouds-bounding-box-detection-and-tracking-pointnet-pointnet-lasernet-67e26116de5a

Reference GitHub Repository:

https://github.com/zhulf0804/PointPillars