Representation, Reconstruction, Recognition

ECE 188: Introduction to Computer Vision

Name: Garvit Rajkumar Pugalia

UID: 504628127

Date: 06/14/2023

Stage 1: Representation

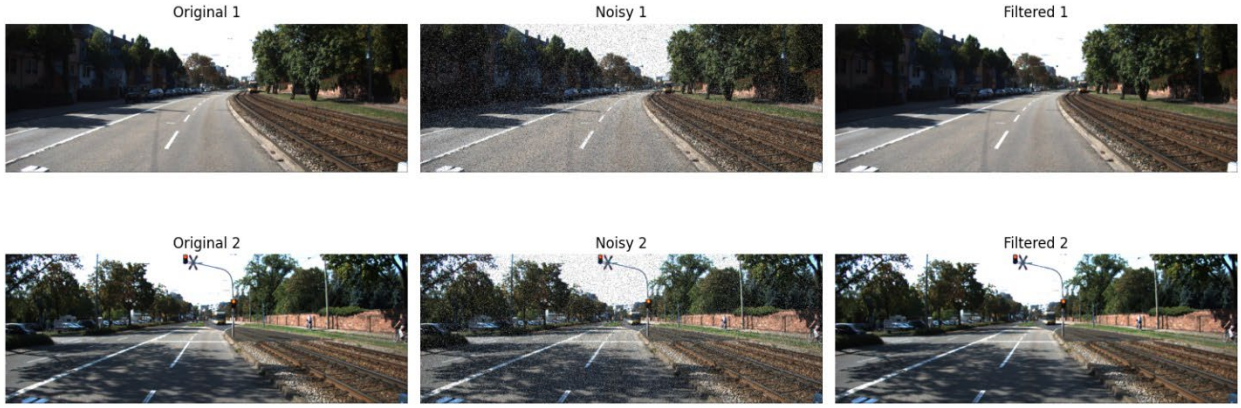
The first stage of the system focuses on removing noise via image processing techniques on the KITTI dataset. The following types of noises were added to the image for testing:

1. Salt and pepper noise
2. Gaussian noise
3. Speckle noise
4. Gaussian blur
5. Gaussian motion blur

These were the methods used and the final peak-signal noise ratio (PSNR) and structural similarity (SSIM) scores for each type of noise. The code can be found here.

1.1 Salt and pepper noise

Since this noise exhibits as white or black pixels, we can use a simple median filter to average out the values of each pixel based on the neighbors and remove these spots in the image. We tune the hyperparameter 'kernel_size' to generate the best result for each of the images. The following figure shows a few examples of the original, noisy, and filtered images.

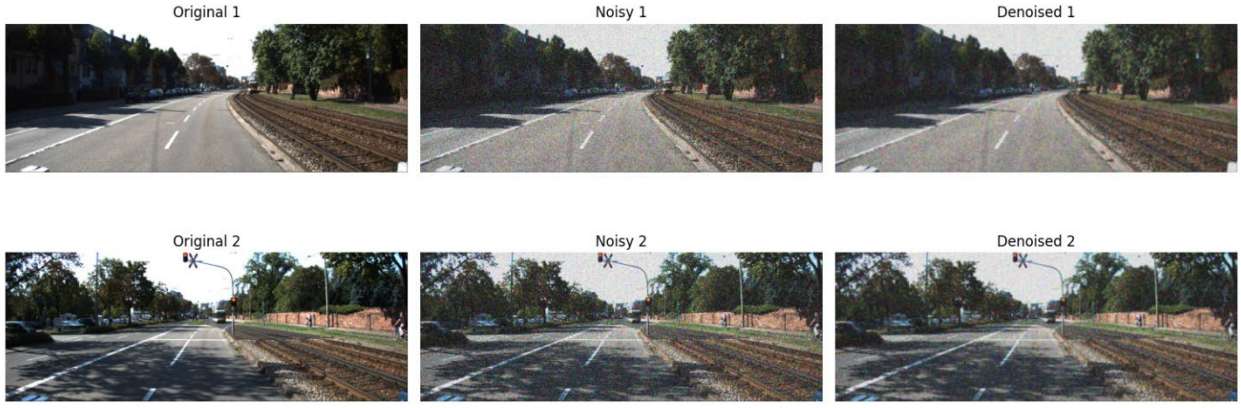


The tunable median filter does a good job at removing the salt and pepper noise with a final score of: *PSNR = 26.77, SSIM = 0.92*.

1.2 Gaussian noise

This is a type of random noise, in this case, modeled as a standard Gaussian added to the image. While this type of noise is tougher to eliminate, one strategy is to apply a Gaussian blur across the image which acts as a low-pass filter to smooth out the 'high frequency' additive signal. We tune the

size of the Gaussian kernel to generate the best result for each of the images. The following figure shows a few examples of the original, noisy, and denoised images.



A more sophisticated denoising method can possibly reduce the noise while maintaining the quality of the image, which is clearly compromised by blurring. However, the method does achieve a final score of: *PSNR = 19.93, SSIM = 0.64*.

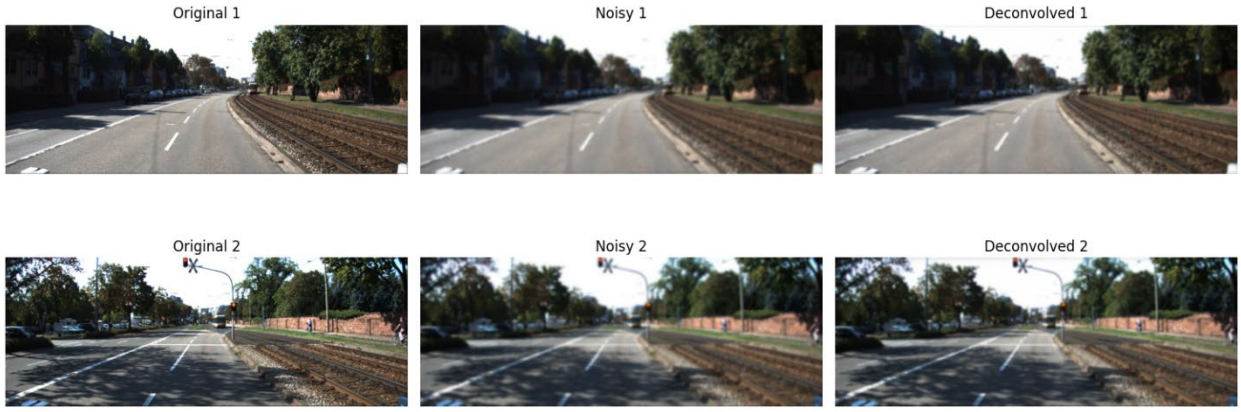
1.3 Speckle noise

This type of noise is similar to salt and pepper, however, it is more expansive and usually creates random patterns of dark and bright spots on the image. Therefore, instead of using a median filter that looks at immediate neighbors, we decided to use a non-local means filter which looks at larger patches on the image for similar valued pixels to average out the noise. [source] We tune the hyperparameters: 'patch_size' which controls the size of patches used for detecting similar pixels, and 'patch_distance' which controls the maximum distance between the pixels being averaged out. The following figure shows a few examples of the original, noisy, and denoised images.

The non-local means filter with tunable parameters does a good job of removing the speckle noise with a final score of: *PSNR = 22.84, SSIM = 0.76*.

1.4 Gaussian blur

As mentioned before, a Gaussian blur operation acts as a low-pass filter and removes high resolution content from the image. A well-known way to remove such blurring and restore the image quality is the Wiener filter. This filter estimates the power spectra of the image to produce an inverse filter for the additive noise in the frequency domain. We tune the size of the point spread function to mimic different sizes of Gaussian blur kernel applied to the image, and a regularization parameter to control the tradeoff between noise reduction and signal preservation. The following figure shows a few examples of the original, noisy, and deblurred images.



One method that we can also try is using an unsupervised Wiener filter, that approximates the regularization parameter automatically based on the image characteristics. However, even with a manually tuned Wiener filter, we get a decent final score of: *PSNR = 21.42, SSIM = 0.72*.

1.5 Gaussian motion blur

We approached the problem with two different algorithms:

- Unsupervised wiener: A version of the Wiener filter where hyperparameters are estimated as part of a stochastic, iterative process.
- Lucy-Richardson algorithm: This is an iterative process to recover the original image by applying a known point spread function to reverse the blurring.

Both algorithms were tuned over an extensive set of hyperparameters. With the unsupervised Wiener method, we were only able to reach a PSNR of 20.06, even after applying more post-processing techniques for resolution such as the Sobel filter. Therefore, we decided to approach the problem in a brute-force method by: (a) tuning the kernel size and angle of the rotated motion blur kernel, and (b) tuning the number of iterations of the Lucy-Richardson algorithm. We achieved this computationally heavy tuning by iteratively reducing the search space around the previously found optimum (for example, start with a 1 step size in angle and check neighbors with a 0.1 step size). The following figure shows a few examples of the original, noisy, and deblurred images for our hyperparameter settings:

Even though this was computationally expensive and took many hours to run, we were not able to reach the target scores; however, we were able to improve the final score to: *PSNR = 20.24, SSIM = 0.68*.

We tried some other methods and post-processing, however, blind deconvolution is a tough problem. As an extra datapoint, we used the original image and noisy image to find the blur kernel in the frequency-domain and used the kernel to de-noise the image. Of course, this is not a blind deconvolution and therefore doesn't help in our problem statement, but it does lead to perfect deconvolution as shown below:

This is a potential path forward to estimate the blur kernel in the frequency domain in a blind manner, and there are a few techniques available online but they are highly complicated and often involve deep learning methods.

Stage 2: Reconstruction

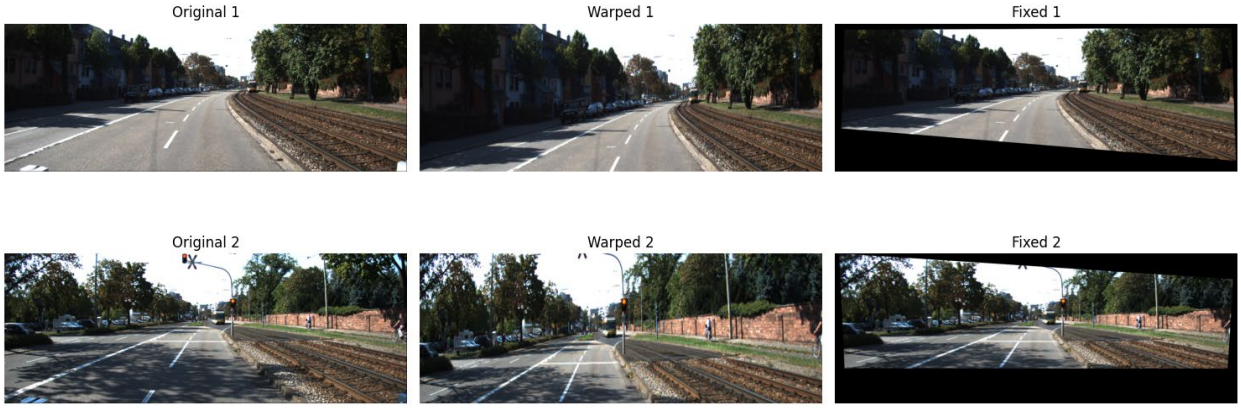
The second stage of the system focuses on reconstructing 3D scenes based on two different camera views of the same scene, thereby exploring topics of rectification and stereo matching. Instead of generating 3D point clouds after stereo matching, we will simply evaluate the results based on ground truth pixel disparity maps. The code can be found here.

2.1 Rectification

For this system, we use the provided input image and warped image to inverse the warping by:

1. Converting the images to grayscale and computing keypoints and descriptors using SIFT.
2. Using a Brute Force feature match algorithm to generate 2 matches in the original image for each keypoint in the warped image.
3. Using Lowe's ratio test to eliminate 'bad' matches. In short, the ratio test removes a keypoint if it's two best matches (from the matching algorithm) are indistinguishable, implying that the keypoint doesn't provide enough information.
4. Using the good keypoint matches to find a homography between the images. Since we get a large amount of matches, we use RANSAC with a reprojection error threshold of 5.0 (to detect inliers) to find a robust homography.
5. Finding the inverse of the homography and applying it to the warped image.

The following figure shows a few examples of the original, warped, and fixed images. The black bars are expected as we lose a certain amount of image when warping.



We could use more sophisticated matching algorithms; however it doesn't have a significant impact with a small number of datapoints. We calculate the final metrics on the largest inscribed rectangle to ignore the black boxes, which gives us a final score of: *PSNR = 31.58, SSIM = 0.92*.

2.2 Stereo Matching

For this system, we are provided images from two stereo cameras, which allows us to estimate the depth and ultimately generate a 3D reconstruction of the scene. To perform stereo matching, we use the OpenCV implementation of the semi-global matching method introduced in this paper.

From my understanding, the method can be divided into four components:

1. Cost: For each pixel in the left image, the method calculates a matching cost (i.e. how well it matches the pixel in the right image) for each value of the disparity in a specified range. We tune this range with the hyperparameter 'numDisparities'. Note: Since the images are rectified, we are matching on horizontal epipolar lines.

2. Aggregation: Instead of relying solely on local data, the algorithm considers global information by aggregating the cost of each pixel along 8 directions (left-to-right, top-to-bottom, etc.) for each value of the disparity. The end result is a cost volume that incorporates the matching cost for the neighborhood into the cost for each pixel and disparity. We use only 5 directions to reduce computational cost.

3. Disparity: The cost volume can be traversed, usually with a straight dynamic programming approach, to find the disparity values with lowest overall cost for each pixel.

4. Post-processing: The method also allows for post-processing methods to check for consistency of the final disparities. A couple methods are mentioned below and were tuned in the process.

Apart from the methods mentioned above, we manually tuned a few parameters over multiple trials to reduce the RMSE.

- numDisparities

- blockSize: This controls the size of the neighborhood over which the matching cost is aggregated. A larger block size can lead to more information captured in the cost volume but can be computationally expensive. A larger block size can also lead to lower resolution and possibly miss subtle depth differences.

- P1, P2: These are penalties associated with large disparity changes between neighboring pixels. These parameters are usually calculated from blockSize, and follow the recommended setting.

- uniquenessRatio: Similar to the concept of Lowe's ratio test, this ratio specifies the difference required in the best and second-best cost for the disparity value to be considered unambiguous.

- speckleWindowSize: Speckles are small regions with significantly different disparity values compared to their neighbors. This noise can be smoothed out by providing the model with the size of possible speckles in the image.

The final disparity maps computed by tuning the StereoSGBM matching algorithm were compared with the ground truth maps with *RMSE = 2.18*.

Stage 3: Recognition

The third stage of this project is an open-ended task on the KITTI 3D dataset, which includes 3D LIDAR point cloud data, 2D stereo images, bounding boxes, labels, camera calibration parameters, and many other data points for common roadway and roadside scenes captured from a vehicle. After exploring the dataset and the task further, it was clear that the most accurate algorithms relied on 3D LIDAR data with many researchers focusing on augmenting the 3D data with features from the other sensors on the car. However, a big problem with LIDAR is that it is expensive as compared to a straightforward stereo camera setup.

With that motivation, we aim to explore two avenues:

- Attempt to understand, train, and evaluate different LIDAR-based models on a subset of the KITTI dataset.
- Attempt to achieve similar success with a stereo-based approach by aligning the stereo embeddings to LIDAR feature maps.

Due to technical limitations, we were unable to fully complete the second task; however, we will discuss the methodology and some of the challenges.

3.1 Background

With the growth of autonomous driving technology, there has been a demand for robust and accurate 3D object detection models, capable of localizing and classifying objects in a scene in real-time. The use of LIDARs in these systems has greatly boosted the effectiveness of object detection, and most of the best-performing models on standard datasets incorporate 3D point cloud data from these sensors in some manner. Furthermore, to handle the sparseness of point cloud data, voxel-based approaches have been commonly used. Analogous to pixels in 2D images, these methods aim to discretize the 3D space into cells to extract important features that can be used in deep learning methods such as convolutional neural networks. One major drawback of these approaches is the memory consumption, which we also faced in our attempts to recreate these methods.

3.2 Data

The KITTI dataset is widely considered as a benchmark for training and evaluating 3D object detection models, as it provides a comprehensive set of real-world data captured by multiple sensors, including stereo cameras and a Velodyne LIDAR. The dataset contains 7,481 training samples and 7,518 test samples, along with a submission portal to evaluate models on the otherwise unannotated testing data.

For the purposes of our exploratory tasks, and due to technical limitations, we used a Python script to sample the KITTI dataset and randomly selected 1000 training and 500 validation point. We used the provided data parser to reduce the 3D point cloud data to match the coordinates of the left stereo image. As part of the training process, we use the provided MMDetection3D APIs to introduce mirroring flips along the X axis, global scaling, and rotation.

3.3 Metric

To evaluate the training and testing performance of the models, we will use the KITTI metric which is based on average precision or AP score. The AP score is calculated based on the IoU (intersection over union) values, which represent the overlap between predicted and ground truth bounding boxes as shown in Figure 1.

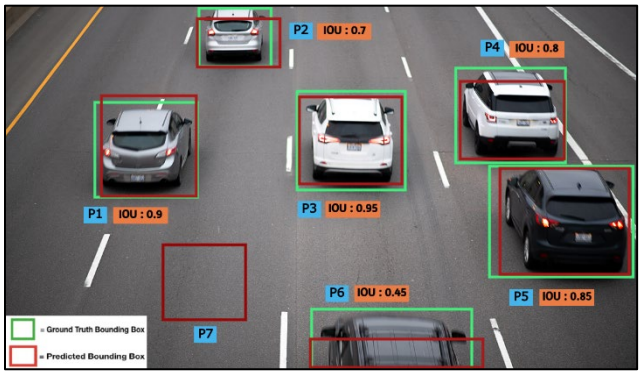


Figure 1: An example of bounding boxes and IoU scoring for 2D object detection.

By setting the threshold of what is considered a true positive detection, we can generate a precision-recall curve. The AP score is the average precision at different recall levels (controlled using IoU thresholds) and is a widely accepted metric for assessment of object detection models. The KITTI metric also provides the benefit of automatically separating the results based on different classes (Car, Pedestrian, Cyclist, etc.) and amount of occlusion in the scene (provided as part of the metadata).

3.4 Methodology

We will first use our data to train two existing LIDAR-based architectures and evaluate the models on our validation and testing sets. Then, we will use a pre-trained model to extract LIDAR feature maps, which will be used with the corresponding stereo images as inputs to build and train a stereo-to-LIDAR sub-model. Due to technical limitations, we train the existing architectures for 40 epochs, increase the learning rate to 0.01, and decrease the batch size to 4.

3.4.1 PointPillars

This architecture was released in 2019, along with pre-trained checkpoints, as a solution to the time efficiency problem of common 3D object detection models. The model accepts 3D point clouds as input, and can be split into three major components (as shown in Figure 2):

- *Pillar Feature Net*: This is an encoder layer that outputs a sparse 'pseudo-image', i.e. a 2D representation of the point cloud data. It first discretizes the point cloud in the x-y plane into 'pillars', while putting constraints on the sparsity and number of pillars. The pillars are then put through a simplified version of PointNet (a unified architecture for various 3D tasks) and maxed over the channel dimension, before being scattered back into their original pillar locations to create a (64, 496, 432) tensor representation.

- *Backbone*: This is a 2D convolution layer that contains sub-networks to extract features at different spatial resolutions, and then concatenate the features by upsampling to a common dimension. Since these use only 2D convolutions, they can be much faster than other architectures that work with 4D tensors.

- *Detection Head*: The final layer is a used for predicting the bounding boxes, and uses a Single Shot Detector (single deep neural network) that relies on default boxes at different aspect ratios and scales to perform matching and regression. The model uses 2D IoU for object matching, and separately provides the box height and elevation as regression targets.

The model is trained on three different loss functions. For classification, most architectures use the focal loss, which is a modified cross-entropy loss to address class-imbalance. For bounding box predictions, they use a smooth L1 loss that incorporates a smoothing term to reduce the influence of outliers. Finally, they use cross-entropy loss for the heading (angle) of the bounding box as it is not accurately captured in the smooth L1 loss.

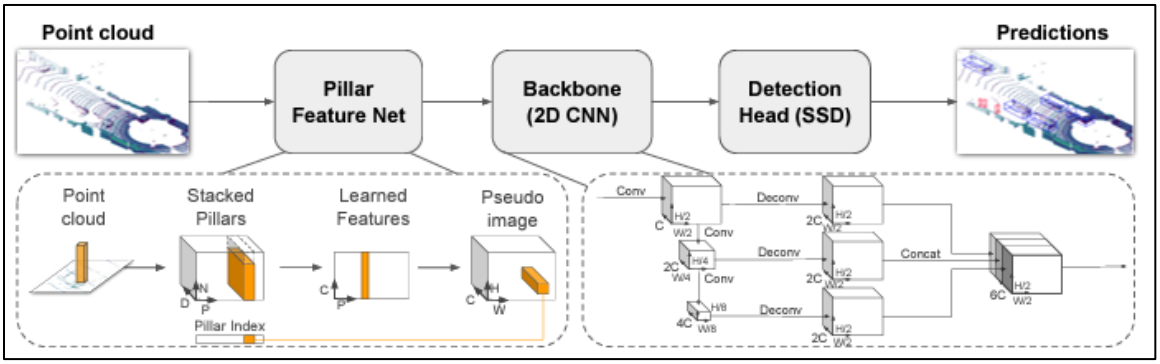


Figure 2: Original architecture of PointPillars model for 3D object detection.

The code for the training can be found here, and the modified configuration files here.

3.4.2 PV-RCNN

The PointVoxel-RCNN, released in 2021, combines a voxel-based convolution model with PointNet-based set abstraction (i.e. size reduction) and thereby takes advantage of the efficiency and quality of both approaches.

As shown in Figure 3, the model converts the 3D point cloud data into voxels of a specified size and applies various 3D sparse convolutions to generate a set of voxel-wise feature vectors which can be used for 3D box proposals. To combat the ineffectiveness and low spatial resolution of 3D feature volumes (such as the one created through voxelization and sparse convolutions), the authors integrate a set abstraction module to: (a) carefully sample a few keypoints from the original point cloud data, and (b) combine the sampled keypoints with neighboring voxels at different levels to generate keypoint features. Now these keypoints have incorporated the voxel-based 3D features and the PointNet-based pointwise features, while preserving the original location information. Finally, these keypoints are aggregated into RoI-grid points for proposal generation and refinement.

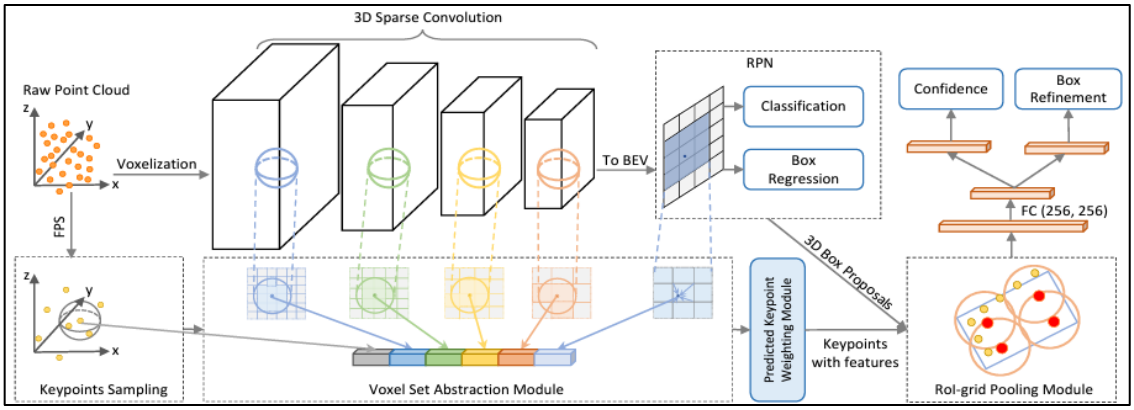


Figure 3: Original architecture of PV-RCNN model for 3D object detection.

Similar to the previous model, PV-RCNN is trained on focal loss for classification labels, L1-smoothing loss for bounding box regression, and cross-entropy loss for the direction of the heading; however, in this case, they are aggregated for three different components in the architecture.

The code for the training can be found here, and the modified configuration files here.

3.4.3 Stereo-based approach

Many approaches such as Stereo R-CNN and DSGN (Deep Stereo Geometry Network) have achieved success in stereo-based object detection; however, these models are unable to reach the performance of LIDAR-based methods. To achieve similar performance as LIDAR with only stereo camera input, we propose 'piggybacking' off the success of a LIDAR-based approach.

As mentioned in the PointPillars method, the encoder layer returns a 'pseudo-image' representation of the 3D point cloud. We use a pre-trained model (on the KITTI dataset) and add a hook to the encoder layer to save the intermediate encoding of the 3D point cloud for our training datapoints. This results in a (64, 496, 432) tensor representation for each datapoint, which acts as the ground truth for our sub-model.

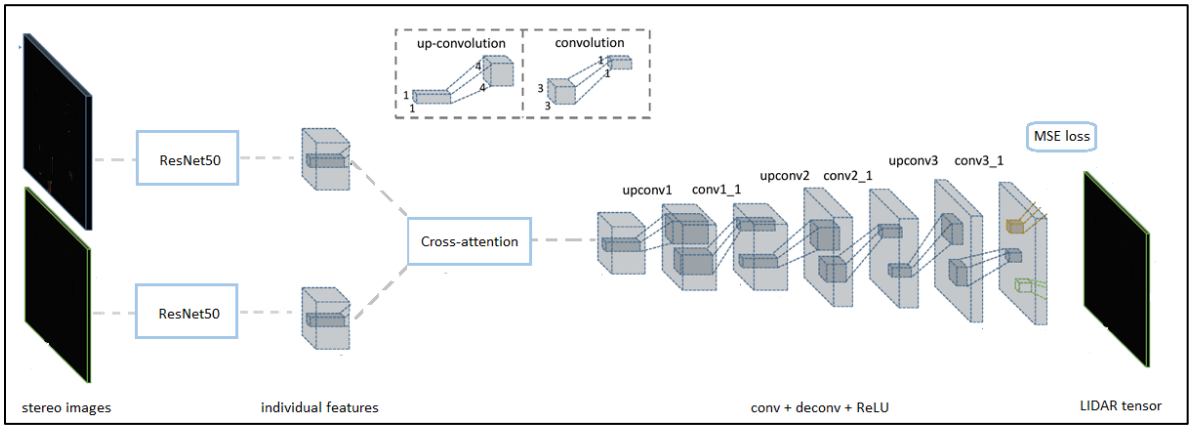


Figure 4: Our proposed sub-model architecture to learn sparse tensor representation of LIDAR-data from stereo image inputs.

For our sub-model architecture, we use pre-trained ResNet50 and remove the last layer to generate feature maps for the left and right stereo images. We concatenate the two images' feature maps along the channel dimension and apply self-attention to add information flow between the two images. Finally, we apply layers of 2D convolution, 2D up-convolution and ReLU to generate a sparse tensor representation matching the LIDAR embeddings. Based on the performance of such an architecture, we can replace the encoder in the PointPillars model to evaluate directly on stereo images.

We were unable to perform any training of this sub-model because: (a) each LIDAR-embedding tensor takes ~52.3MB, (b) we import pre-trained weights for the ResNet50 model, and (c) we are working with convolutions on large tensors, which cannot be effectively used with our GPU constraints.

The code for extracting intermediate layer embeddings can be found here, and the unsuccessful attempt to train the sub-model here.

3.5 Results

The following table shows the results of the two LIDAR-based approaches for 2D and 3D bounding box estimation on the validation dataset.

Model	**Type**	**AP11**			**AP40**		
		easy	mod.	hard	easy	mod.	hard
PointPillars	2D	66.23	57.52	54.22	66.80	57.30	53.40
PV-RCNN		75.88	66.68	63.66	77.60	67.77	64.24
PointPillars	3D	42.47	37.02	34.03	40.84	33.98	31.51
PV-RCNN		51.17	44.97	41.77	49.59	42.51	39.44

With 40 epochs, the PV-RCNN model achieves better validation accuracy than the PointPillars model across all categories. We wrote the following notebook to visualize a few datapoints for different threshold values of the score (i.e. the confidence level of the prediction) and the ground truth bounding boxes.

Figure 5: Ground truth annotations for test images

The ground truth boxes for the two images display the cars, tram, and pedestrian. In our case, a tram is considered a vehicle and should be identified by the model.

Figure 6: Predicted bounding boxes from PV-RCNN with high confidence, threshold = 0.7.

There are some positives and negatives in the results from the PV-RCNN model. It can detect objects with higher confidence than PointPillars. However, it looks like there is more training required as it is unable to identify the tram or correctly draw the bounding box around the pedestrian. These are datapoints that are less frequent in the KITTI dataset, and it is possible that there were not enough samples to learn from.

Figure 7: Predicted bounding boxes from PointPillars for threshold = 0.5 (left) and threshold = 0.7 (right)

For the PointPillars model, the visualization matches the empirical results of the training and evaluation. First, it has lower confidence for correctly identified objects as seen by the extra car in the results with a lower threshold. Second, it fails to identify any objects in our second example i.e. the pedestrian. This can be attributed to lack of samples in the training set, however, it still performs worse compared to the PV-RCNN model.

3.6 Conclusion

PV-RCNN improves on the convolutional voxel-based approach, which PointPillars uses with a slight modification, by incorporating point-based techniques that have proven to work empirically. Therefore, it is not surprising that our results indicate the same superiority in performance. However, as mentioned before, PointPillars can get relatively close AP scores with around ~1 hour of training, compared to ~6 hours taken by PV-RCNN. This displays a key tradeoff in 3D object detection between approaches that attempt to reduce the complexity of the model by relying on 2D convolutions and clever dimensionality reductions (PointPillars) versus approaches that try to boost performance by computing and incorporating more features in the architecture (PV-RCNN).

For the second part of our project, where we try to align the embeddings of stereo images to the 'pseudo-images' extracted from PointPillars, there were many technical limitations. In fact, the embeddings had to be extracted from PointPillars in batches of 500 to avoid memory issues, as each batch of tensors equated to ~25GB of data. While the efforts were futile, we want to optimize our model going forward and test out the intuition to understand if stereo images can indeed mimic the LIDAR-based approach and if embedding alignment can improve existing stereo-based models.

3.7 References

KITTI data: https://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=3d

PointPillars: https://arxiv.org/pdf/1812.05784.pdf

PV-RCNN: https://arxiv.org/pdf/1912.13192.pdf

SSD: https://arxiv.org/pdf/1512.02325.pdf

PointNet: https://arxiv.org/pdf/1612.00593.pdf

Inspiration for UpConv architecture: https://arxiv.org/pdf/1411.5928.pdf

Some modifications made to toolkit: https://github.com/garvitpugalia/mmdetection3d

All code, notebooks, and data:
https://drive.google.com/drive/folders/1kY471nn6keuCucke9wiHfCfoDRio5xLS?usp=sharing