DenseFuseNet: Improve 3D Semantic Segmentation in the Context of Autonomous Driving with Dense Correspondence

Yulun Wu
Shanghai Jianping High School
lunw1024@gmail.com

Abstract

With the development of deep convolutional networks, autonomous driving has been reforming human social activities in the recent decade. The core issue of autonomous driving is how to integrate the multi-modal perception system effectively, that is, using sensors such as lidar, RGB camera, and radar to identify general objects in traffic scenes. Extensive investigation shows that lidar and cameras are the two most powerful sensors widely used by autonomous driving companies such as Tesla and Waymo, which indeed revealed that how to integrate them effectively is bound to be one of the core issues in the field of autonomous driving in the future. Obviously, these two kinds of sensors have their inherent advantages and disadvantages. Based on the previous research works, we are motivated to fuse lidars and RGB cameras together to build a more robust perception system.

It is not easy to design a model with two different domains from scratch, and a large number of previous works (e.g., FuseSeg[10]) has sufficiently proved that merging the RGB camera and lidar models can attain better results on vision tasks than the lidar model alone. However, it cannot adequately handle the inherent correspondence between the RGB camera and lidar data but rather arbitrarily interpolates between them, which quickly leads to severe distortion, heavy computational burden, and diminishing performance.

To address these problems, in this paper, we proposed a general framework to establish a connection between lidar and RGB camera sensors, matching and fusing the features of the lidar and RGB models. We also defined two kinds of inaccuracies (missing pixels and covered points) in spherical projection and conducted a numerical analysis on them. Furthermore, we proposed an efficient filling algorithm to remedy the impact of missing pixels. Finally, we proposed a 3D semantic segmentation model, DenseFuseNet, which incorporated our techniques and achieved a noticeable 5.8 and 14.2 improvement in mIoU and accuracy on top of vanilla SqueezeSeg[24]. All code is already open-source on https://github.com/IDl0T/DenseFuseNet.

Keywords: *Autonomous driving, 3D semantic segmentation, lidar, point clouds, sensor fusion, spherical projection, noise analysis, convolutional neural networks*

Contents

1 Introduction **3**

2 Related Work **6**
2.1 Semantic Segmentation on RGB Images 6
2.2 Instance Segmentation on RGB Images 6
2.3 Semantic Segmentation on Point Clouds 6

3 Our Method **9**
3.1 Spherical Projection 9
3.2 Missing Pixels and Covered Points 10
3.3 DenseFuseNet Framework 11
3.3.1 Filling Missing Pixels 11
3.3.2 Hidden Layer to Input Layer 13
3.3.3 Input Layer to Input Layer 14
3.3.4 Warp and Fuse the Feature 15
3.4 Network Architecture 15
3.5 Complexity Analysis 15

4 Experiments **15**
4.1 Dataset 16
4.2 Evaluation Metrics 17
4.3 Quantitative Analysis 18
4.4 Training Method 18
4.5 Dense Correspondence Effectiveness 20

5 Conclusion **21**

A Lidars **23**

B Featured Source Code **24**
B.1 Model Backbone 24
B.2 Data Pipeline 31

1 Introduction

Autonomous vehicles are a fascinating field that is completely changing the nature of our lives. On the one hand, it is expected to solve a crucial cause of mortality (i.e., traffic accidents) and save 1.35 million lives per year [13]. On the other hand, it releases new sources of productivity and creates new avenues for growth, especially for physical disabilities. By 2020, the global market demand for autonomous vehicles is estimated to be approximately 67,000, and the compound annual growth rate from 2021 to 2030 is expected to reach 63.1% [8]. However, due to the inability to ensure absolute safety (i.e., accurately perceive the surrounding environment), modern autonomous systems still cannot be widely applied [1, 22].

We compared Lidar and RGB cameras from several aspects. Autonomous vehicles typically carry a combination of sensors (i.e., lidar, and RGB camera[1]) to perceive the surrounding environment as Figure 2. As shown in Table 1, Lidar and cameras have their own shortcomings. First, lidars can provide accurate depth information with point clouds, but it can also be easily obscured by external conditions such as rain and fog. Meanwhile, although the camera outputs high-resolution images, as a "dumb" sensor, it only produces planar images (i.e., 2D images) and loses sight without light. Since these two sensors can complement each other, designing a general fusion model will undoubtedly be the future direction of autonomous vehicles.

Table 1. Comparison of RGB camera and lidar sensors.

	RGB camera	Lidar
Output Format	2D images	3D point cloud
Resolution	High	Low
Depth Information	Not directly available	Can be precisely retrieved
Environmental Requirements	Good illumination	Clean air (Fog and raindrops absorb lasers)
Price	$50-$1000	$10k

Although the benefit of sensor fusion is apparent, most previous work only using lidar point clouds [2, 12, 24, 25, 26], while ignoring the complementary information from RGB cameras. A recent work named FuseSeg [10] proposed several methods to fuse features between the lidar and the RGB camera. Typically, these approaches utilize the spherical projection [24]) to transform the 3D point cloud into the 2D representation (i.e., lidar image). For FuseSeg, the farthest point sampling and first-order spline interpolation are applied to guide the fusion of the lidar and the RGB camera. In summary, these methods have the following problems, 1) The noise in the point cloud is ignored when applying spherical projection; 2) Since the sparse position of the feature map is interpolated during the fusion process, harmful distortion and artifacts are naturally generated; 3) The time complexity of the existing method is exceptionally high. It is hardly scalable with increasing resolution; 4) Since FuseSeg do not provide source code, it is challenging to reproduce its fusion method.

To address these problems, we propose an innovative framework for sensor fusion, named DenseFuseNet. Our goal is to utilize the RGB camera to improve segmentation accuracy on 3D semantic segmentation. As shown in Figure 1, 3D semantic segmentation is to perceive the surrounding environment in the form of lidar point clouds as Section 2.3. For spherical projection noises, we first divide the

[1]Radars are minor sensors that are not included in our research.

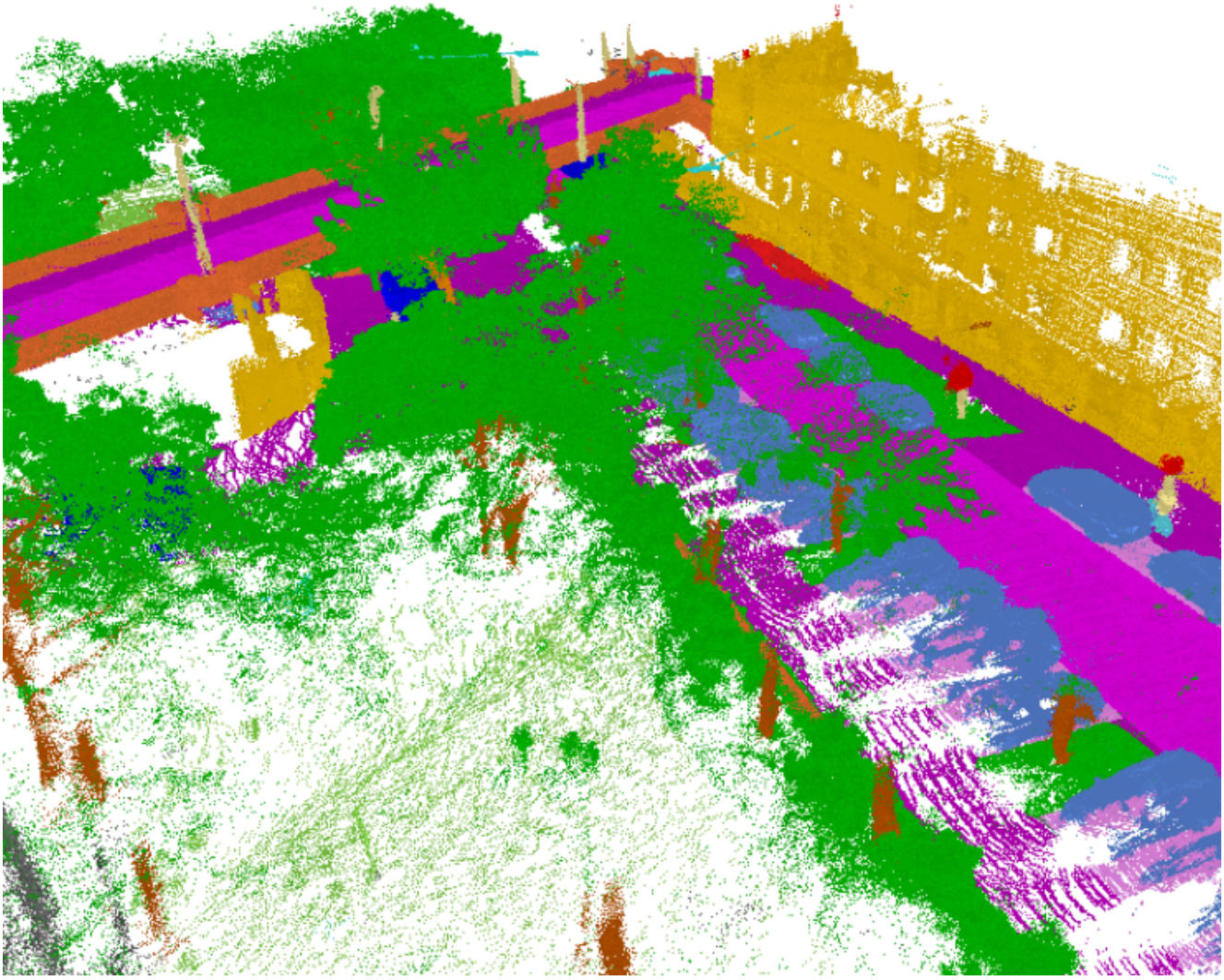


Figure 1. An example result of open space 3D semantic segmentation. Blue: car, purple: ground, green: vegetation, yellow: building.

noise into two categories: missing pixels and covered points. Afterward, we conducted a quantitative analysis of the two kinds of noises to evaluate their impact on 3D semantic segmentation. As far as we know, we are the first to analyze the defects of spherical projection. It is worth noting that we also have proposed an efficient GPU-based algorithm to resolve missing pixels[2] as Section 4.3. For the rest of the problems, we exploit the calibration data between sensors and spatial invariance of CNNs to establish a

[2]Because the KNN post-processing method can adequately resolve the cover points, we will not discuss them in our research.

dense and accurate feature mapping relationship between arbitrary layers of the lidar branch (SqueezeSeg [24]) and the RGB branch (pretrained MobileNetv2 [19]). Then, we warp and fuse the features of the pretrained MobileNetv2 [19] into SqueezeSeg according to the correspondence between their feature maps. Our method has lower time complexity compared with FuseSeg[10] and generate more accurate feature correspondences to obtain better sensor fusion quality. Moreover, our method can be directly applied to many of the state-of-the-art 3D semantic segmentation models. We evaluated DenseFuseNet on the 3D semantic segmentation dataset SemanticKITTI [3], which contains 23,201 pairs of lidar scans and RGB images. Compared with the original SqueezeSeg[3], our DenseFuseNet achieves a noticeable improvement of 5.8 and 14.2 in mIoU and accuracy, respectively. On the other hand, DenseFuseNet performs inference at 23 FPS[4], surpassing the 20 FPS sample rate of typical lidars.

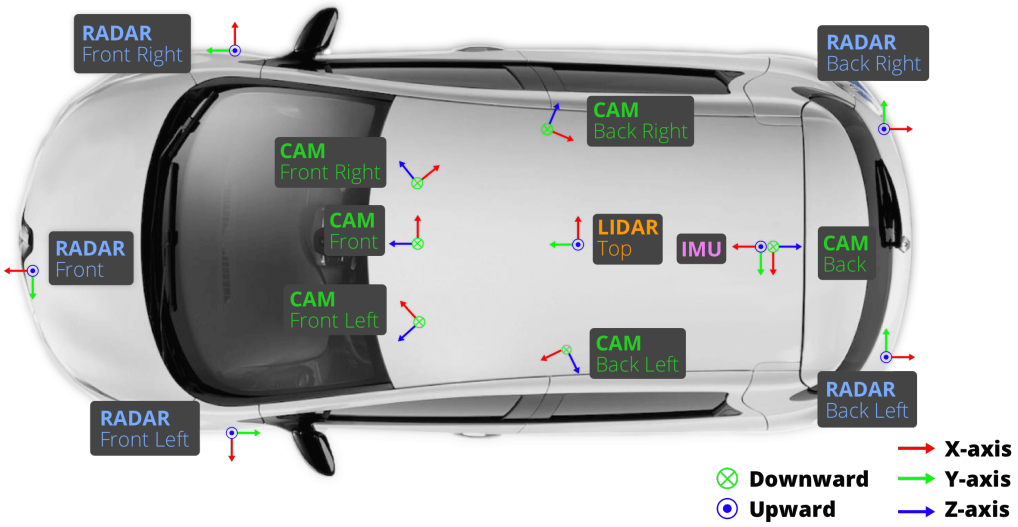


Figure 2. An standard sensor suite of an autonomous vehicle from nuScenes [4].

In summary, our main contributions are:

- We quantitatively analyzed the impact of noise on spherical projection and proposed an effective GPU-based filling algorithm to solve the missing pixel problem.
- We proposed a highly generalizable framework named DenseFuseNet to establish dense feature correspondences for the fusion between the lidar and RGB camera sensors. Moreover, it can be generalized to most of the state-of-the-art 3D semantic segmentation models.
- We evaluated DenseFuseNet on a large-scale multi-sensor dataset SemanticKITTI and achieved a significant improvement of 5.8 and 14.2 in mIoU and accuracy over baseline.
- We published our training, analysis and visualization code on GitHub: https://github.com/IDl0T/DenseFuseNet

[3]Because the original SqueezeSeg used a different dataset, we reproduced SqueezeSeg by training it from scratch.
[4]FPS means frame per second.

Our proposed method has great significance in both academics and industry. In academics, we are the first to utilize quantitative analysis to reveal the defects in spherical projection and comprehensively resolve the multi-sensor fusion. We provide a reliable benchmark and open code base for other researchers for further research. In the industry, autonomous vehicle companies can fine-tune and adapt our methods to build robust and accurate perception systems by fusing the lidar and RGB camera. Our research is a milestone in the field of autonomous driving and provides a stable framework for multi-sensor fusion.

2 Related Work

In this section, we will briefly introduce the previous research works for the semantic and instance segmentation on RGB images and point clouds, respectively.

2.1 Semantic Segmentation on RGB Images

Semantic segmentation on RGB images is a computer vision task that is used to label each pixel of an image with a corresponding category (such as car, road, cars, pedestrians, cyclists, etc.). A sample of semantic segmentation is shown in Figure 3(a). There are a lot of research works on semantic segmentation [11, 18, 23]. FCN [11] is one of the earliest innovative semantic segmentation methods based on deep neural networks. It used transposed convolutional layers to upsample extracted image features and restore pixel category from encoded features in an end-to-end way. For more accurate semantic segmentation, U-Net [18] is proposed to apply an encoder-decoder architecture. It used shortcut connections to crop and concatenate early feature maps with the feature maps after upsampling, enabling the feature and gradient to better propagate through the entire model. The state-of-the-art semantic segmentation approach is HRNet [23], which maintains high-resolution representations throughout the network, and does parallel information exchange between multiple resolution representations.

2.2 Instance Segmentation on RGB Images

Semantic segmentation means recognizing objects by their categories, while instance segmentation takes a step further to assign different instances. An obvious comparison of semantic segmentation and instance segmentation is shown in Figure 3. One of the most characteristic work in instance segmentation is Mask R-CNN [9], which combines a object detection model, Faster-RCNN [17], and FCN [11]. It used a region proposal network [7] to locate regions of interest and utilize them for further classification, bounding box regression, and mask generation. Compared with semantic segmentation, instance segmentation is more challenging and usually requires computationally heavy models like Mask R-CNN, therefore being less suitable for real-time applications.

2.3 Semantic Segmentation on Point Clouds

Different from semantic segmentation on RGB images, semantic segmentation on point clouds means assigning a class label to every point in a 3D point cloud as shown in Figure 1. 3D semantic segmentation can be conducted on closed scenes (such as the room) or open space (such as the highway). In the closed scenes, the input point clouds are usually of small size and simple categories, so that it is less demanding on network efficiency. In contrast, the point clouds of open scenes are massive and complicated and the

(a) Semantic segmentation (b) Instance segmentation

Figure 3. A comparison between semantic segmentation and instance segmentation. Instance segmentation distinguishes different instances of the same class, but semantic segmentation does not.

density of clouds can also significantly change at different distances. It's worth noting that we focus on Open space 3D semantic segmentation in this project.

Similar to semantic and instance segmentation on RGB images, instance segmentation on point clouds can distinguish different objects of the same category. Although the instance segmentation can provide more information for the autonomous driving system, it usually requires models with huge parameters and cannot perform inference in real-time because of its higher speed and intensive computation requirements. At present, many work on autonomous driving regard semantic segmentation as an essential task of autonomous driving, and many datasets (such as SemanticKITTI [3]) only provides semantic labels rather than instance annotations. Besides, semantic segmentation can be converted to instance segmentation by clustering methods [15]. Therefore, our work mainly focuses on the semantic segmentation of point clouds.

In summary, the previous works of semantic segmentation on point clouds can be divided into four directions. In the following subsections, we will introduce these directions in detail.

1) *Semantic segmentation straight on the point clouds* means that no preprocessing and feature representation is performed on the point clouds. This series of methods treated point clouds as an unordered point array. For instance, PointNet [16] uses a series of multi-layer perceptrons to process the point array and operates a global max pooling to retrieve an invariant feature to the input permutation. However, it only extracts global features, which is inconsistent with the designation of modern CNNs (extracting local features per layer). To obtain the local features of different layers, PointNet++ [16] proposes a multi-level network structure to extract hierarchical features. However, these methods are still incredibly slow since they need to handle sparse and unordered data, which is hard to accelerate. Therefore, it cannot reach the real-time inference speed that autonomous driving requires.

2) *Semantic segmentation using spherical projection* embeds the 3D point clouds into 2D representations (the mechanics behind spherical projection are listed in Section 3.1). The idea of spherical projection was first introduced in SqueezeSeg [24]. It can convert the sparse data structure (i.e., point clouds) into a dense 2D representation (i.e., lidar image). This process permits the use of convolutional neural networks, which are faster and more parameter-efficient than other systems dealing with 3D point clouds directly. Following works like SqueezeSegv2 [25] and SqueezeSegv3 [26] further enhanced SqueezeSeg to boost the performance. SqueezeSegv2 [25] introduced a Context Aggregation Module (CAM) to deal with the noise in the projected lidar image, and SqueezeSegv3 [26] combined the Spatially Adaptive Convolution (SAC) to apply adaptive filters for different locations of the input lidar image. In other words, it considers the different nature of lidar images in various image regions. SqueezeSegv3 [26]

is one of the best-performing models in 3D semantic segmentation, but it compromises its speed down to around 10 FPS[5]. Aside from the SqueezeSeg series, many other works such as RangeNet++[12] also adopt spherical projection as a preprocessing step. In practice, due to external causes like transparent material and precision error, noises can appear in the projected lidar image, which can significantly reduce the performance.

It is worth noting that there are some attempts to utilize neural networks to learn a more accurate 2D representation from the 3D point cloud to solve the noise in spherical projection. One of the most typical work is 3D-MiniNet [2], which uses the Convolutional Neural Networks (CNNs) to convert the original 3D point clouds into 2D representations and combine them with lidar images from spherical projections. As far as we know, 3D-MiniNet [2] is one of the highest performing models in 3D semantic segmentation, which fully illustrates the necessity of our research. We are the first research work to analyze point cloud noise and its impacts in detail. More importantly, our GPU-based filling algorithm can effectively solve the missing pixel problem.

3) *Semantic segmentation with multi-sensor fusion* attempts to utilize multiple sensors to solve the noise of point clouds caused by the lidar. One recent work, FuseSeg [10], improved the performance of SqueezeSeg [24] by fusing it with a pretrained image classification model: MobileNetv2 [19]. FuseSeg [10] performs farthest point sampling, which has a time complexity of $O(n^2)$[6], on the lidar point clouds to select a set of "control points" on the lidar feature, calculating their corresponding positions on the RGB feature. Based on that, it also uses first-order spline interpolation ($O(n^2)$ to establish a sub-pixel correspondence to fuse the RGB feature to the lidar.

Figure 4. Illustration of warping artifacts produced by FuseSeg [10]. There are noticeable ghosting and distortion in the image, such as van roof and cyclist.

[5]FPS means frames per second.

[6]n is the number of points.

In general, FuseSeg [10] has several drawbacks: 1) The noise in the point clouds is ignored when applying spherical projection; 2) It can cause severe distortion as shown in Figure 4, mainly due to interpolation on a small number of "control points". At the fusion stage, only 48 "control points" are selected out of 32,000+ total points per lidar scan; 3) FuseSeg utilizes farthest point sampling and first-order spline interpolation, both of which have a time complexity of $O(n^2)$. The high time complexity causes its inference time to increase dramatically (empirically proved in [10]) as the number of control points goes up. Manually reducing the number of control points is only a temporary solution because it will hinder us from dealing with higher resolution sensors. On the other hand, reducing the number of control points will also bring more vital disturbance and seriously affect performance. 4) FuseSeg [10] did not publish its source code so that it is difficult to reproduce and compare. Compared with FusionSeg, our proposed method provides an open-source and reliable codebase to fuse RGB cameras and lidars. It is a foundation work in sensor fusion and can apply to almost state-of-the-art 3D semantic segmentation methods.

3 Our Method

In this section, we first review the definition of spherical projection to reveal its noise problem. Based on that, we defined two types of spherical projection noises (i.e., cover points and missing pixels) and quantitatively analyzed their impact on 3D semantic segmentation. Then, we introduced our proposed multi-sensor fusion framework, which contains four steps and can effectively fuse lidar and RGB cameras. Finally, we presented our network structure and analyzed its time complexity.

3.1 Spherical Projection

Many previous works on 3D semantic segmentation [12, 24, 25, 26] rely on spherical projection. It is a common approach to project 3D point clouds into 2D representations (i.e., liar image) that CNNs can process. As shown in supplementary material A, because the lidar has the constant horizontal and vertical angular resolution, spherical projection can naturally embed point clouds. In other words, when the position of the lidar is set as the origin of the coordinates, the spherical projection can theoretically reconstruct 3D point clouds into a gridded 2D lidar image.

Formally, spherical projection is to project a set of 3D points $[x, y, z]$ to pixels of lidar images $[u, v]$ by:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} h \cdot (1 - (\arcsin(z/\sqrt{x^2+y^2+z^2}) + \text{fov}_{\text{down}})/\text{fov}) \\ w \cdot \frac{1}{2}(1 - \text{atan2}(y, x)/\pi) \end{bmatrix} \tag{1}$$

where $[x, y, z]$ represents an input point in point clouds, and $[u, v]$ is the 2D coordinate of the lidar image. h and w are the height and width of the output lidar image. Note that $\text{fov}_{\text{up/down}}$ is the upper/lower field of view of the lidar, and $\text{fov} = \text{fov}_{\text{down}} + \text{fov}_{\text{up}}$.

However, the spherical projection itself has apparent defects. Due to the inherent system deviation of the lidar sensor (especially on the vertical axis projection), the spherical projection sometimes projects multiple lidar points (i.e., $[x, y, z]$) onto the same lidar image pixel (i.e., $[u, v]$), while some lidar image pixels end up with no lidar points projected on it. According to our analysis in Section 4.3, these projection errors will reduce the segmentation accuracy by 2.4%. Therefore, how to quantify and reduce

their impact is an essential issue in our research. In the next section, we will analyze the two kinds of noise caused by projection error in detail.

3.2 Missing Pixels and Covered Points

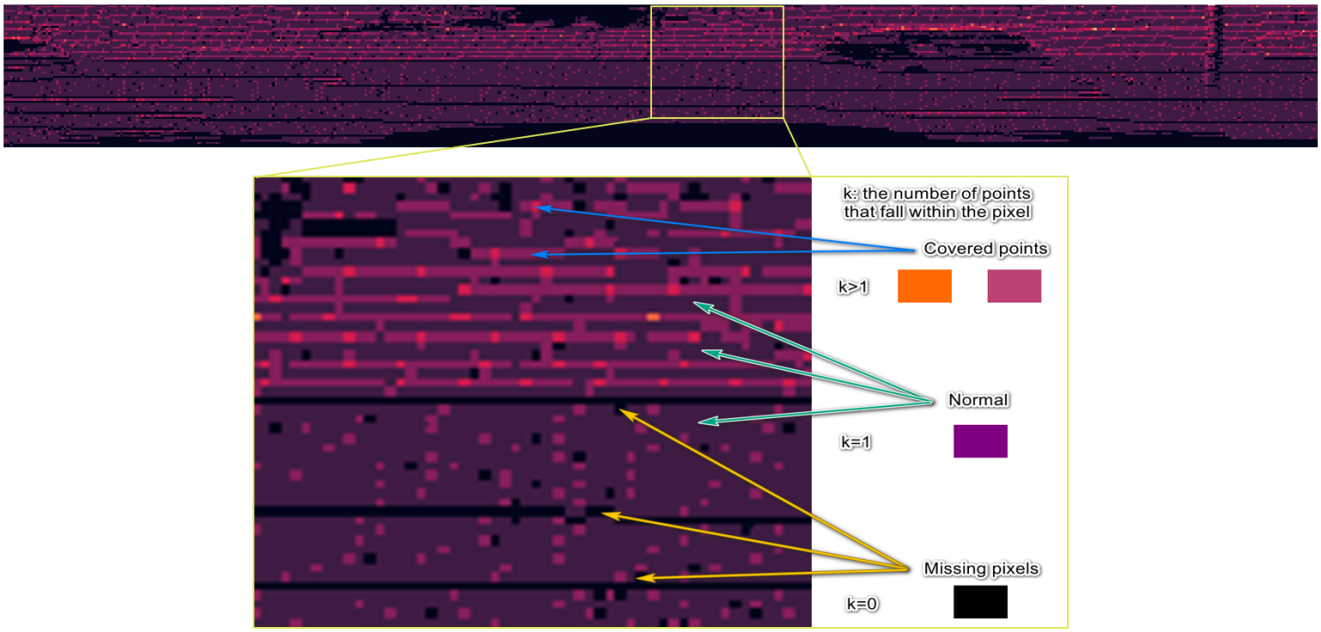


Figure 5. The frequency map indicating how many lidar points are projected onto a pixel. The black area where frequency $= 0$ denotes missing pixels, while brighter pixels where frequency > 1 contains (frequency $- 1$) covered points. Pixels where frequency $= 1$ (dark purple areas) are normal situations.

To solve the projection error, we define two types of noise (i.e., *missing pixels* and *covered points*) caused by spherical projections on the lidar image. An example of missing pixels and covered points are in Figure 5.

Missing pixels are the pixels in the projected lidar image that end up with no lidar points projected on it. In other words, it also implies that many pixels in range images cannot trace back to their three-dimensional coordinates in 3D lidar points. Missing pixels can be classified into three types: 1) the "pepper and salt" missing pixels caused by the inherent noise; 2) the stripe-shaped missing pixels caused by the structural inaccuracy (the inaccuracy in pitch angle) of the lidar; 3) the large missing patches mainly caused by surfaces that hardly reflect lasers like car windows and the sky.

Covered points appear when multiple lidar points are projected into the same lidar image pixel due to that one lidar image pixels can only hold the information of one lidar point. Covered points will introduce two problems: 1) a part of the useful input information is lost; 2) some covered points might be misclassified[7]. Both of them can seriously impact the segmentation accuracy. Problem one is generally

[7]The segmentation CNN outputs a segmented lidar image while we want a segmented point cloud. If we label the points in the original point cloud by the class of the lidar image pixel that they are projected onto, and this could lead to misclassification where lidar points of different class overlap in a pixel. For example, a part of a pedestrian behind a car might be misclassified as car class.

inevitable, but according to PointNet[8] [16], it cannot have too much impact. Problem two could be remedied by CRF [27] or KNN post-processing [12], so we did not include this problem in our research. We still quantified impact of covered points in Section 4.

3.3 DenseFuseNet Framework

In this section we introduce our sensor fusion framework, named DenseFuseNet. Our goal is to fuse the RGB camera and the lidar. Therefore, we attempted to find a correspondence between the feature maps of two models. To better explain our method, we first introduce the concept of dense correspondence. We formally define a dense correspondence f as a mapping between feature maps:

$$\forall(i_A, j_A) \in \mathbf{A}, (i_A, j_A) \xrightarrow{f} (i_B, j_B), s.t.(i_B, j_B) \in \mathbf{B} \tag{2}$$

where **A**, **B** are rectangular feature maps, and (i, j) denotes pixel coordinates. For clarity, a dense correspondence is a pixel to pixel mapping between two feature maps.

Once we have got a dense correspondence from **A** to **B**, we can warp and fuse **B** to **A** by concatenating by channels. The result will be a feature map of the same size as **A**, with as many additional channels as **B**. In order to perform a reliable feature fusion, we should fuse features by their relationship in space. For example, we should fuse the region of **B** that contains a car to the region of **A** that contains the same car. Only so the feature fusion would be meaningful.

To perform sensor fusion, we propose a 4-step framework (Figure 6): 1) Fill the missing pixels in the lidar image; 2) Establish the dense correspondence between hidden layers and input layer of either branch; 3) Establish a dense correspondence from lidar image to RGB image (i.e., input layers of two branches); 4) Combine the dense correspondence for a correspondence between feature maps of two models, and use it to warp and fuse RGB features into the lidar branch. In the following four subsections, we will explain the detail of each step in our framework.

3.3.1 Filling Missing Pixels

We propose an efficient filling algorithm to fill missing pixels mentioned in Section 3.2. In the early phase of our research, we first tried to use a masked median filter combined with heuristics to fill the entire lidar image. We tried to fill missing pixels by the following steps: 1) For every missing pixels p_0, select the set of pixels $s = \{p|D(p, p_0) < k\}$ where $D(x, y)$ denotes the Manhattan distance between pixels x and y. Smaller k results in finer filling result, but less missing pixels are filled. 2) Fill p_0 with the median in s. 3) Fill the rest of the points with heuristics specific to channels (range, x, y, z, reflectance). For example, fill the top half of the range channel with the global maximum, and fill the lower half with the local minimum. However, this algorithm was hard to compute in parallel since the size of s is variable.

To solve this problem, we proposed a new GPU-based filling algorithm. Our algorithm applies a series of median filters with increasing kernel sizes on the lidar image. When applying each filter, we use a mask to track the missing pixels left to fill and always keep the original data outside the mask. In other words, we only fill the missing parts and keep the rest untouched. Our algorithm has a decent time complexity of $O(H \times W \times \sum \frac{k_i^2}{s_i^2}) = O(n)$, where k_i and s_i are the kernel size and stride for i-th

[8] PointNet [16] proved that a small subset of points is enough to characterize an object.

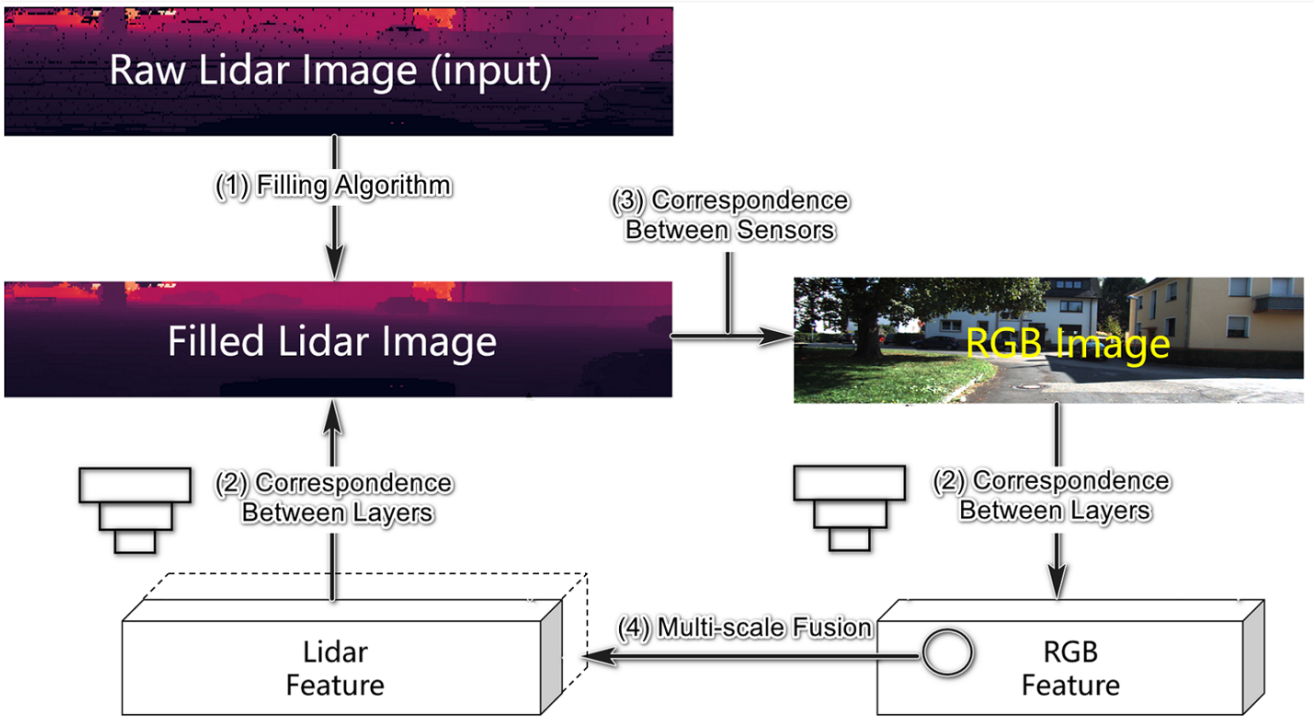


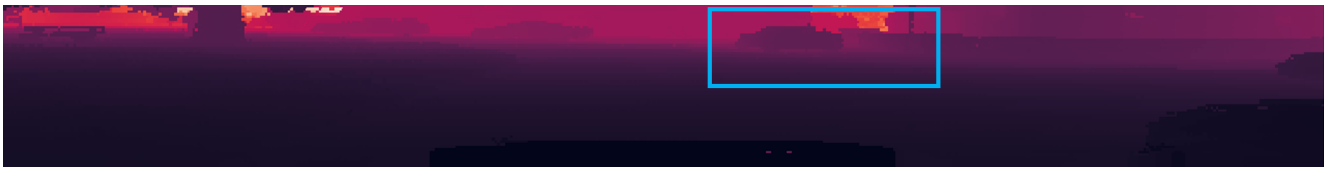
Figure 6. An overview of the 4-step framework to establish dense correspondence between feature layers. (1) fills the missing pixels in the input lidar image; (2) denotes the dense correspondence within branches established by space invariance; (3) is the dense correspondence between input layers established by lidar-camera calibration; (4) combines (2) and (3) to infer correspondences between multiple feature layers, and uses those correspondences to warp and fuse features.

filter. The overall time complexity is constant because the kernel size is much less than image resolution, and we only use 4-5 filters in total. Moreover, since our algorithm consists of purely median filters, it can be massively accelerated with a GPU, making it even faster. With this configuration, our proposed method does not increase any observable cost in inference speed. It is worth noting that our algorithm successfully reduced the percentage of missing pixels from 24.323% to 6.274%, as described in Section 4.3. More details of our implementation are listed in Algorithm 1.

Algorithm 1: Filling missing pixels

Input: lidar image, initial mask
Output: filled lidar image
filters $\leftarrow$ 3x3, 5x5, 7x7, 13x13, ...; `// median filters of different sizes`
for filter in filters **do**
    median $\leftarrow$ filter(lidar image);
    lidar image $\leftarrow$ lidar image + median $\times$ mask;
    mask $\leftarrow$ where abs(lidar image) $<$ eps; `// pixels still missing`

(a) Unfilled r channel



(b) Filled r channel

(c) Zoomed

Figure 7. Filling the r (range) channel of a lidar image. (a) is the unfilled r channel, (b) is filled with our algorithm, and (c) is a zoomed comparison between them.

3.3.2 Hidden Layer to Input Layer

In this section, we calculate the dense correspondence from hidden layers to the input layer. In CNNs, a pixel on the feature map is a convolution sum over an usually rectangular receptive field. In light of this, we calculate the dense correspondence between layer $l+1$ and layer l. For every pixel in the feature map of layer $l+1$, we map it to the geometric center of its receptive field as Figure 8 and Equation 3.

$$A_{l+1}[i,j] \mapsto A_l[\lfloor \frac{k_h+1}{2} \rfloor + i \cdot s_h, \lfloor \frac{k_w+1}{2} \rfloor + j \cdot s_w] \tag{3}$$

where $[i,j]$ are image coordinates and k_h, k_w, s_h, s_w are kernel sizes and strides on each direction. In this process, we use the geometric center of the receptive fields of the feature map as the representative position for each pixel on the feature map of layer $l+1$, and we can repeat the process to calculate the correspondence between arbitrary hidden layers to the input image of the network.

With this operation, we can calculate the correspondence as long as we are able to determine the "center." Thus, our method is highly generalizable for most space-invariant models. For instance, one state-of-the-art model of 3D semantic segmentation, 3D-MiniNet [2], is also applicable by our method since it's based on grid neighbor searches. In this case, we can turn 3D-MiniNet into a sensor fusion model in split seconds.

Specific to the model backbones in this paper, they are only using regular 1x1 pad 0 and 3x3 pad 1 convolutions. Thus, we can derive a clean formula for the dense correspondence between a-th layer and

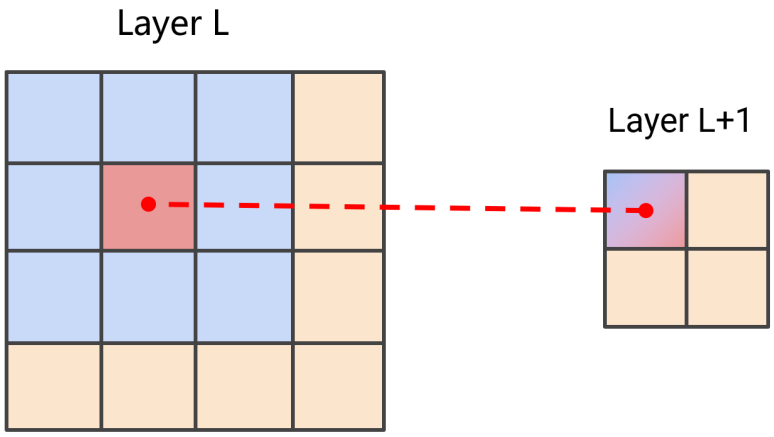


Figure 8. The correspondence between layer $l+1$ and layer l, with a 3x3 kernel. Even kernel size also works by introducing floating number.

b-th layer, where $a < b$, for the lidar branch SqueezeSeg[9]:

$$A_b[i,j] \rightarrow A_a[i, j \times \text{downsample factor}] \tag{4}$$

and for the RGB branch MobileNetv2:

$$A_b[i,j] \rightarrow A_a[\lfloor \frac{i}{\text{downsample factor}} \rfloor, \lfloor \frac{j}{\text{downsample factor}} \rfloor] \tag{5}$$

The correspondence is constant for a certain network architecture, so we can calculate it in advance and use hash map to access. The time complexity for this step is $O(n)$, where n is the number of pixels in the feature map.

3.3.3 Input Layer to Input Layer

In this section, we establish connections between input layers of lidar branch and RGB branch (i.e., lidar image and RGB image). The best way to establish this connection is to exploit the calibration file usually provided by a multi-sensor dataset. The calibration file specifies a 3x4 matrix $\mathbf{P}$, which denotes the transformation from the lidar coordinate to RGB camera coordinates. Therefore, we are able to map the points in the lidar point cloud to pixels on the RGB image:

$$\mathbf{x}_{\text{rgb}i} = \mathbf{P}_i \mathbf{Tr} \mathbf{x}_{\text{lidar}} \tag{6}$$

where $\mathbf{Tr}$ is the transformation matrix from the lidar coordinate to the 0-th camera coordinate. $\mathbf{P}_i$ is the rectified[10] transformation matrix from 0-th camera coordinate to the i-th camera coordinate. For example, applying $\mathbf{Tr}$ and $\mathbf{P}_2$ translates points in the lidar coordinate $\mathbf{x}_{\text{lidar}} = [x, y, z, 1]^T$ to pixel positions on the second camera $\mathbf{x}_{\text{rgb}} = [r, c, 1]^T$.

[9] SqueezeSeg only downsamples on width.

[10] Rectification means making image planes from different cameras co-planar.

3.3.4 Warp and Fuse the Feature

We combine the dense correspondences in Section 3.3.2 and 3.3.3 for the correspondence f between feature maps of lidar branch and RGB branch. Given f, we are able to fuse feature maps together. We warp the RGB features such that $A_{\text{warped}}[i,j] = A_{\text{rgb}}[f(i,j)]$, and simply concatenate A_{warped} to the lidar model as extra channels. There are more possible ways to fuse features, but it is out of the scope of this paper and might be covered in future works.

3.4 Network Architecture

We adopted the model structure of FuseSeg[10], which constitutes two branches: a SqueezeSeg [24] as the segmentation backbone and a MobileNetv2 [19] as the RGB feature extractor. We used the implementation of MobileNetv2 from PyTorch Hub [14] with the pretrained weight on ImageNet [5]. To show that it is RGB images rather than more parameters that improved segmentation, we did not fine-tune the MobileNetv2, even though it should improve performance further. The feature maps of the 7th, 14th, 19th layers of a MobileNetv2 are fused to the fire2, fire4, fire7 layers of SqueezeSeg by simply concatenating them by channel. We choose the similar-sized feature maps just before down-sampling for fusion to retain the most information.

3.5 Complexity Analysis

In this section, we analyzed the time complexity of our proposed method. To facilitate understanding, we gradually explained the time complexity of each step in our framework as:

- Filling missing pixels : $O(n)$ (median filters)
- Calculate correspondence from hidden layer to input layer: $O(n)$ (using hash table)
- Calculate correspondence between input layers of two branches: $O(n)$ (using calibration matrix)
- Warp and fuse the features: $O(n)$

where n = resolution of the lidar image = number of points in the point cloud. It is worth noting that the time complexity of our method ($O(n)$) significantly surpasses FuseSeg[10] ($O(n^2)$), which fully demonstrates that our approach can efficiently fuse the lidar and RGB camera.

4 Experiments

In this section, we first introduce the SemanticKITTI [3] dataset and compare it with its predecessor, the KITTI [6] dataset. Then we conducted a quantitative analysis of missing pixels and covered points. Finally, we introduced the training method and verified the effectiveness of our proposed 4-step framework as in Section 6.

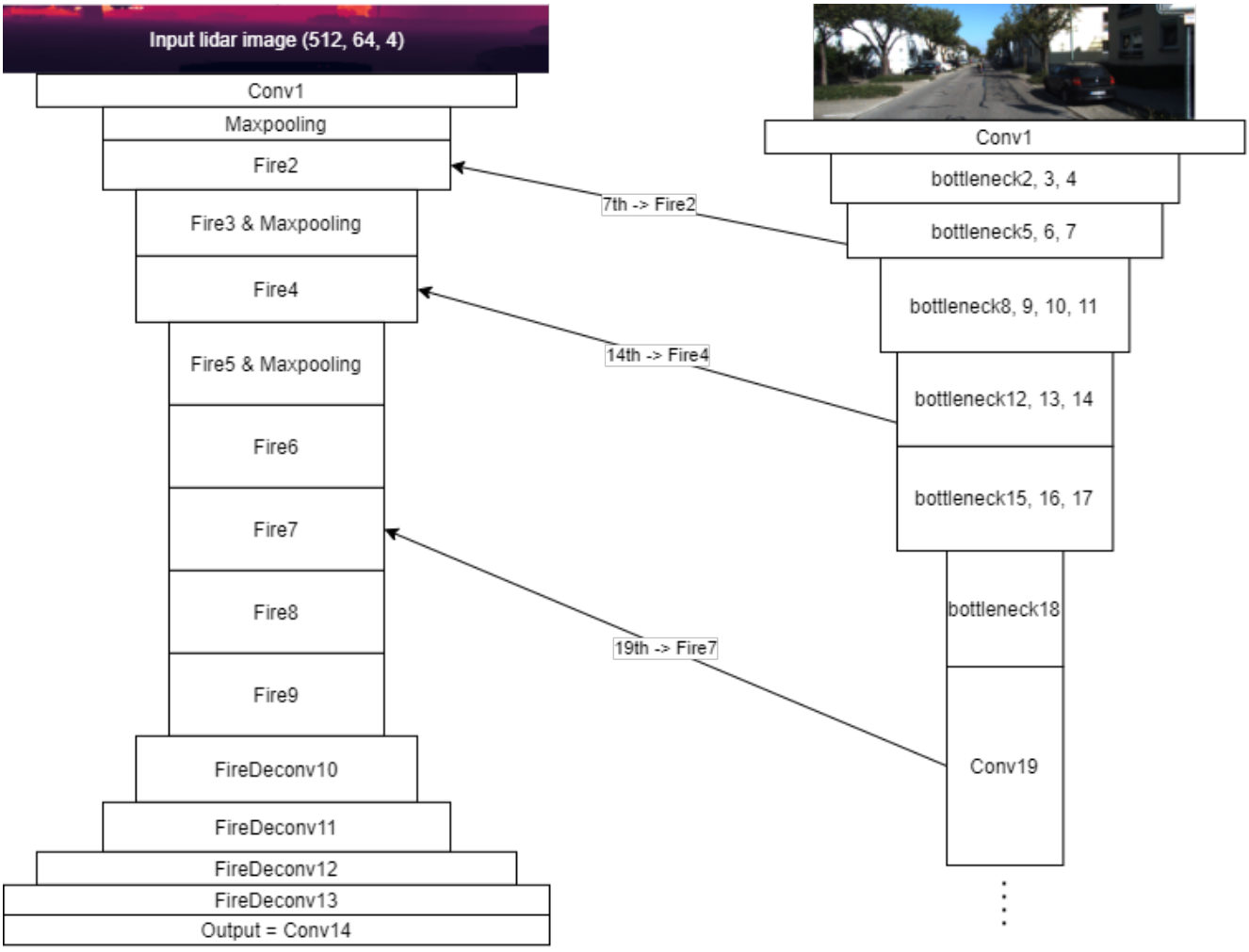


Figure 9. The architecture of DenseFuseNet.

4.1 Dataset

Different from many previous works [24, 10], which are trained and evaluated on KITTI [6], we used SemanticKITTI [3] dataset in this paper. SemanticKITTI is the largest lidar-based semantic segmentation dataset with 23,201 and 20,351 scans in the train set and test set, and 4,549 million points in total. Compared with KITTI, SemanticKITTI provides point-wise annotation, an improvement on the 3D bounding box annotation of KITTI. More and more recent works such as SqueezeSegv3 [26] and 3D-MiniNet [2] turns to SemanticKITTI instead of KITTI.

However, SemanticKITTI does not provide an official benchmark for sensor fusion models. So, we did some modifications on the original dataset. First, we cropped the 360° lidar scan to the front 90° **to match the camera field of view**. Second, we aligned the left RGB camera images to the lidar scans and bundled them together as our model's input.

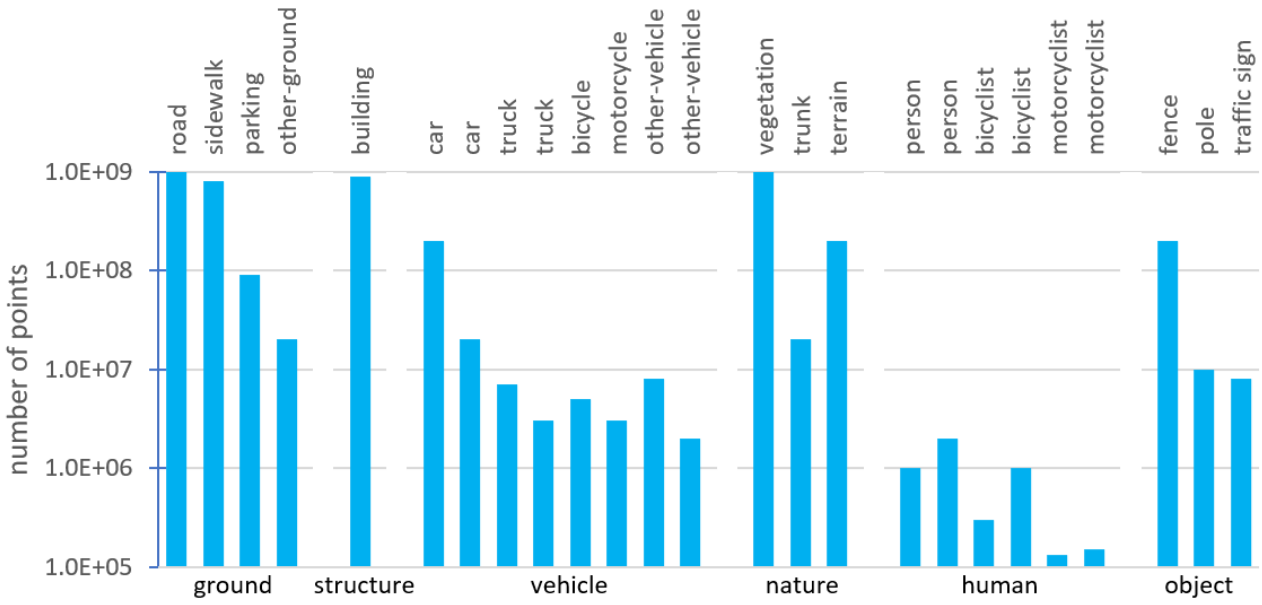


Figure 10. The label distribution of SemanticKITTI [3].

4.2 Evaluation Metrics

We designed some metrics to evaluate the impact of missing pixels and covered points 3.2:

$$\text{Missing percentage} = \frac{\text{missing pixels}}{\text{total pixels}} \times 100\% \tag{7}$$

$$\text{Covered percentage} = \frac{\text{covered points}}{\text{total points}} \times 100\% \tag{8}$$

$$\text{Percentage of CPF} = \frac{\text{CPF}}{\text{covered points}} \times 100\% \tag{9}$$

$$\text{Accuracy drawback} = \frac{\text{CPF}}{\text{total points}} \times 100 \tag{10}$$

Where CPF means the number of covered points falsely classified if no post-processing was performed. That is to say, the final result is calculated by projecting the segmented lidar image back to the point cloud, using predicted class of pixels to label the entire point cloud. In this process, some points can be misclassified. Specifically, we calculates CPF by two steps. First, we project the ground truth label into a segmented lidar image. Then, we use the segmented lidar image to label the point cloud again and count how many points are misclassified (only covered points could be misclassfied in this setting).

To evaluate 3D semantic segmentation, we used the common metrics in semantic segmentation, mean Intersection over Union (mIoU) and Accuracy:

$$\text{mIoU} = \frac{1}{k+1} \sum_{i=0}^{k} \frac{TP}{FN + FP + TP} \tag{11}$$

where $k+1$ = number of classes.

$$\text{Accuracy} = \frac{TP + TN}{FN + FP + TP + FP} \tag{12}$$

4.3 Quantitative Analysis

We conducted a statistical analysis on SemanticKITTI to better understand the impact of missing pixels and covered points on segmentation. More details of missing pixels and covered points are listed in Section 3.2 We sampled a 140-million-point subset from SemanticKITTI to speedup the process. Ideally, since we set the resolution of the lidar image the same as the resolution of the lidar, the number of missing pixels should be equal to the number of covered points. In other words, every missing pixel must result in an extra point projected into another pixel. However, the number of covered points is less than missing pixels in practice, since some lasers emitted by the lidar never come back. Formally, $S + C = W \times H$, where S is the size of the point cloud and C is the number of laser beams. Our experimental results are listed in Table 2.

Table 2. Statistical Analysis on SemanticKITTI

Quantity	Value
Missing pixels	36,191,641
Covered points	27,932,231
Total points	140,540,078
Total pixels in lidar image	148,799,488
Missing percentage	24.323%
Missing percentage after filling	6.274%
Covered percentage	19.878%
CPF (covered point falsely classified)	3,329,974
CPF percentage	11.923%
Accuracy drawback	2.369

The effects of missing pixels are listed in the table 2. We observed that the lidar image was not as dense as we thought. As shown in the table 2, 24.323% of the lidar image pixels are missing. After using our filling algorithm (Algorithm 1), we successfully reduced the pixel loss percentage to 6.274%. This fully illustrates that our approach can generate dense lidar images that are more suitable for CNNs. Regarding the effect of covered points, our statistics show about 19.9% points are covered points. In other words, spherical projection causes a 19.9% information loss. Moreover, if no post-processing is applied, 11.9% of these points will be misclassified even if the lidar images are perfectly segmented. In reality, this could lead to an accuracy drawback of at least 2.4%.

4.4 Training Method

Our model was trained with PyTorch 1.5.0 [14] on 4 NVIDIA Titan X GPUs for 70 epochs in about 48 hours. We built our model on top of the Lidar-bonnetal framework [12]. When we implemented and trained DenseFuseNet for the first time using the same learning rate as SqueezeSeg, we found that all training metrics (loss, accuracy, iou) started to deteriorate after some epochs, as shown in Figure 11.

Table 3. Mean IoU (intersection over union) and accuracy of our model and baseline. The highest score in each column is marked in bold.

Method	mIoU (%)	Accuracy (%)
SqueezeSeg	17.0	59.7
DenseFuseNet	**22.8**	**73.9**
Difference	+5.8	+14.2

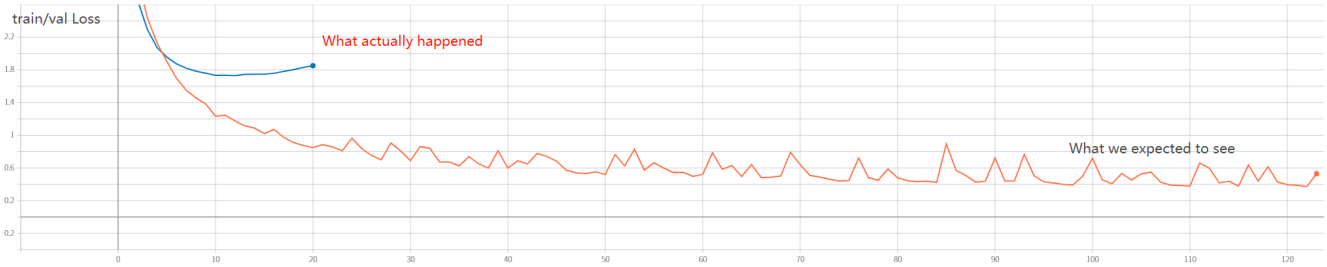


Figure 11. The yellow curve is the training loss curve of SqueezeSegv3 [26], and the blue curve is what happened using the Lidar-bonnetal framework [12].

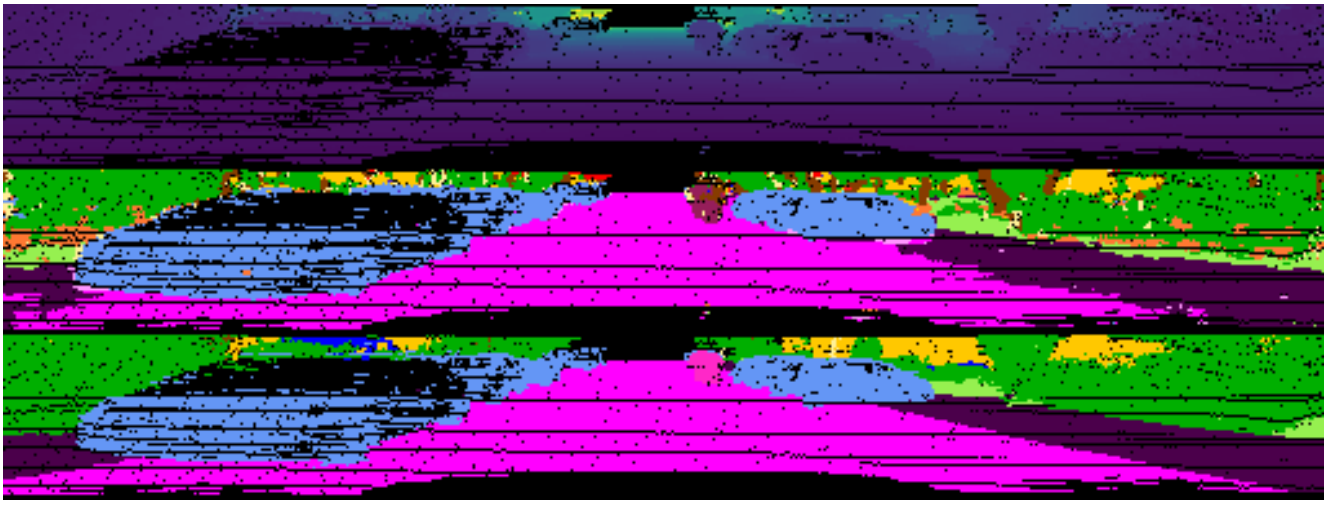
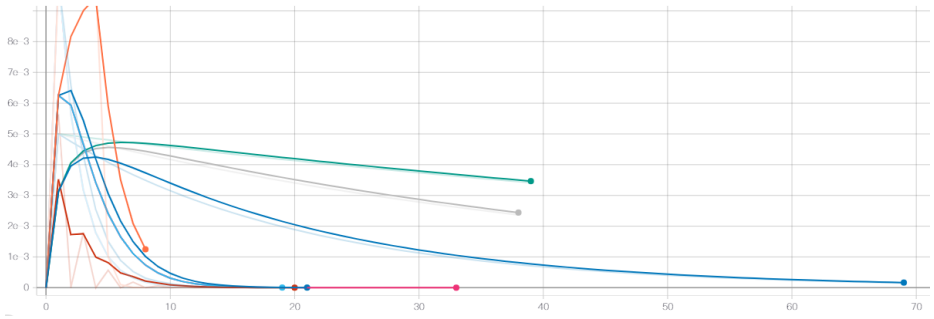


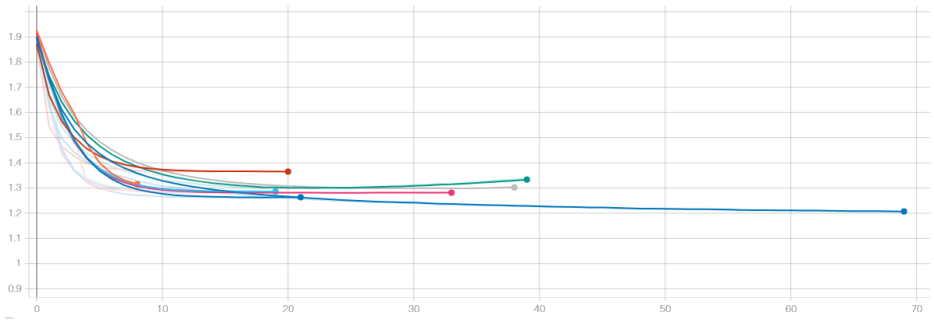
Figure 12. Segmentation result of DenseFuseNet. From top down is respectively depth image, prediction, and ground truth.

This seems to be a common problem due to the ill-conditioned Hessian matrix, which could be alleviated by increasing momentum and reducing the learning rate. We tried several strategies [20, 21] to adjust the learning rate. However, the peak performance didn't improve much, as in Figure 13(c). We also trained a vanilla SqueezeSeg [24] from scratch using the lidar-bonnetal framework.

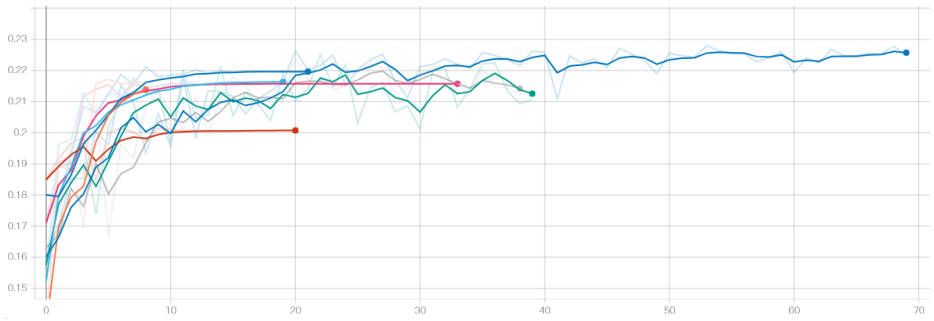
Again, the training metrics started to deteriorate after 20 epochs, and the performance peaked at iou=0.17, much lower than the peak iou claimed by the framework author. To locate the problem, we trained from scratch a SqueezeSegv3 [26] using another framework, and it behaves just as the paper claimed. Thus, we speculate that a bug causes this issue on training in the framework [12], but we were unable to locate it. We have contacted the author and will continue to work on this problem. Still, we compared DenseFuseNet and vanilla SqueezeSeg as Table 3. Our method achieved a noticeable improvement of 5.8 in mIoU and 14.2 in accuracy over the reproduced SqueezeSeg. A sample segmentation



(a) Learning rate



(b) Training loss



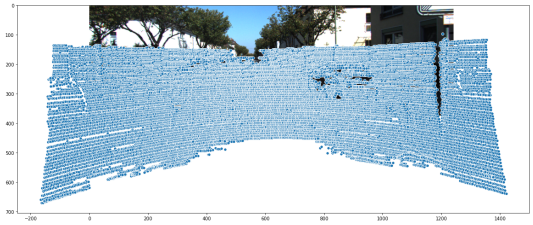
(c) Valid IoU

Figure 13. The training logs of DenseFuseNet with different learning rate scheduling and momentum.

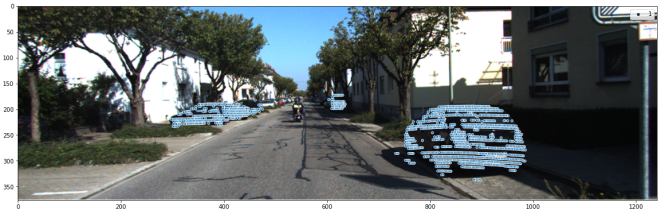
result on the validation set is described in Figure 12.

4.5 Dense Correspondence Effectiveness

We tested our correspondence computation method as Section 3.3. Specifically, we calculated a dense correspondence between lidar and RGB images directly for visualization purposes. We projected the lidar points to their corresponded position on RGB images. The results are shown in Figure 14. We can see that the lidar points are projected onto the RGB image with high accuracy. On top of that, we tried to warp the RGB image into a "ready-to-fuse" form as Section 3.3.4. From the Figure 15, we can observe that our method significantly improves the performance of FuseSeg [10]. Compared with FuseSeg, our approach can effectively reduce the disturbance around the object boundary.



(a) All points in the point cloud



(b) Points of class “car”

Figure 14. The visualization of the calculated dense correspondence between the input layers as in 3.3.3. Lidar points are projected to its corresponded counterpart.

(a) Original image

(b) Warped image by dense correspondence

Figure 15. The image warped according to the calculated dense correspondence between the input layers. It doesn’t have the ghosting in Figure 4. The color inaccuracy is due to the dataset we used.

5 Conclusion

In this paper, we proposed a general and efficient framework to guide sensor fusion. We first quantitatively analyzed the impact of two kinds of noises caused by spherical projection on 3D semantic segmentation. Then we designed a GPU-friendly algorithm to handle the noises in projected lidar images. Finally, we proposed a 3D semantic segmentation model, named DenseFuseNet, to fuse a SqueezeSeg and a MobileNetv2. On the SemanticKITTI dataset, our DenseFuseNet achieved a noticeable improvement of 5.8 in mIoU and 14.2 in accuracy over the original SqueezeSeg. Our method provides a baseline for sensor fusion in 3D semantic segmentation, opening up a novel pathway fusing fully-fledged models into a multi-input model. We believe our work will empower the multi-sensory perception system of autonomous vehicles and hopefully accelerate the popularization of autonomous vehicles.

References

[1] ABC7NEWS. Tesla self-driving car fails to detect truck in fatal crash. https://abc7news.com/tesla-s-autopilot-self-driving-car-officials-investigating-teslas-autopilot-feature-after-fatal-crash/1410042/. 3

[2] I. Alonso, L. Riazuelo, L. Montesano, and A. C. Murillo. 3d-mininet: Learning a 2d representation from point clouds for fast and efficient 3d lidar semantic segmentation. arXiv preprint arXiv:2002.10893, 2020. 3, 8, 13, 16

[3] J. Behley, M. Garbade, A. Milioto, J. Quenzel, S. Behnke, C. Stachniss, and J. Gall. SemanticKITTI: A Dataset for Semantic Scene Understanding of LiDAR Sequences. In Proc. of the IEEE/CVF International Conf. on Computer Vision (ICCV), 2019. 5, 7, 15, 16, 17

[4] H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom. nuscenes: A multimodal dataset for autonomous driving. arXiv preprint arXiv:1903.11027, 2019. 5

[5] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In CVPR09, 2009. 15

[6] A. Geiger, P. Lenz, and R. Urtasun. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR), pages 3354–3361, 2012. 15, 16

[7] R. Girshick. Fast r-cnn. In Proceedings of the IEEE international conference on computer vision, pages 1440–1448, 2015. 6

[8] I. Grand View Research. Autonomous vehicle market size, share & trends analysis report by application (transportation, defense), by region (north america, europe, asia pacific, south america, mea), and segment forecasts, 2021 - 2030. Technical Report GVR-4-68038-401-7, Grand View Research, Inc., 2020. 3

[9] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask r-cnn. In Proceedings of the IEEE international conference on computer vision, pages 2961–2969, 2017. 6

[10] G. Krispel, M. Opitz, G. Waltner, H. Possegger, and H. Bischof. Fuseseg: Lidar point cloud segmentation fusing multi-modal data. In The IEEE Winter Conference on Applications of Computer Vision, pages 1874–1883, 2020. 1, 3, 5, 8, 9, 15, 16, 20

[11] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 3431–3440, 2015. 6

[12] A. Milioto, I. Vizzo, J. Behley, and C. Stachniss. RangeNet++: Fast and Accurate LiDAR Semantic Segmentation. In IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS), 2019. 3, 8, 9, 11, 18, 19

[13] W. H. Organization. Road traffic injuries. https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries. 3

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In Advances in neural information processing systems, pages 8026–8037, 2019. 15, 18

[15] Q.-H. Pham, T. Nguyen, B.-S. Hua, G. Roig, and S.-K. Yeung. Jsis3d: joint semantic-instance segmentation of 3d point clouds with multi-task pointwise networks and multi-value conditional random fields. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 8827–8836, 2019. 7

[16] C. R. Qi, H. Su, K. Mo, and L. J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 652–660, 2017. 7, 11

[17] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In Advances in neural information processing systems, pages 91–99, 2015. 6

[18] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In International Conference on Medical image computing and computer-assisted intervention, pages 234–241. Springer, 2015. 6

[19] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 4510–4520, 2018. 5, 8, 15

[20] L. N. Smith. Cyclical learning rates for training neural networks. In 2017 IEEE Winter Conference on Applications of Computer Vision (WACV), pages 464–472. IEEE, 2017. 19

[21] L. N. Smith and N. Topin. Super-convergence: Very fast training of neural networks using large learning rates. In Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications, volume 11006, page 1100612. International Society for Optics and Photonics, 2019. 19

[22] J. Stewart. Why tesla's autopilot can't see a stopped firetruck. https://www.wired.com/story/tesla-autopilot-why-crash-radar/. 3

[23] J. Wang, K. Sun, T. Cheng, B. Jiang, C. Deng, Y. Zhao, D. Liu, Y. Mu, M. Tan, X. Wang, et al. Deep high-resolution representation learning for visual recognition. IEEE transactions on pattern analysis and machine intelligence, 2020. 6

[24] B. Wu, A. Wan, X. Yue, and K. Keutzer. Squeezeseg: Convolutional neural nets with recurrent crf for real-time road-object segmentation from 3d lidar point cloud. In 2018 IEEE International Conference on Robotics and Automation (ICRA), pages 1887–1893. IEEE, 2018. 1, 3, 5, 7, 8, 9, 15, 16, 19

[25] B. Wu, X. Zhou, S. Zhao, X. Yue, and K. Keutzer. Squeezesegv2: Improved model structure and unsupervised domain adaptation for road-object segmentation from a lidar point cloud. In 2019 International Conference on Robotics and Automation (ICRA), pages 4376–4382. IEEE, 2019. 3, 7, 9

[26] C. Xu, B. Wu, Z. Wang, W. Zhan, P. Vajda, K. Keutzer, and M. Tomizuka. Squeezesegv3: Spatially-adaptive convolution for efficient point-cloud segmentation. arXiv preprint arXiv:2004.01803, 2020. 3, 7, 9, 16, 19

[27] S. Zheng, S. Jayasumana, B. Romera-Paredes, V. Vineet, Z. Su, D. Du, C. Huang, and P. H. Torr. Conditional random fields as recurrent neural networks. In Proceedings of the IEEE international conference on computer vision, pages 1529–1537, 2015. 11

A Lidars

Lidars (Figure 16) are arguably the most important sensors in autonomous driving. A lidar emits pulsed laser beams into its surroundings and measured their reflections to calculate the travel distance of each laser beam and the reflectance of the surface the laser beam hits by measuring reflection intensity. Formally, a laser beam emitted by the lidar corresponds to one point $[x, y, z, r]$ in space, where x, y, z is its coordinate in the 3D space, and r is the reflectance of the material it hits. Figure 1 shows an example of lidar point clouds. Notice that the density of point clouds decreases as distance increases.

Lidars are widely adopted by various automotive corporations like Waymo, Lyft, Baidu, etc., as the primary sensor for their autonomous driving system. Lidars can be used in various ways to perceive

Figure 16. A Velodyne HDL-64E lidar sensor.

surroundings, such as 3D object detection, point cloud segmentation, and SLAM. Lidars are useful in many fields like robotics, geography, and autonomous driving. Therefore, it is valuable to develop methods based on lidars.

B Featured Source Code

B.1 Model Backbone

```
from __future__ import print_function
import math
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn.modules.utils import _pair, _quadruple

class Fire(nn.Module):
  """
    In channel: inplanes
    Out channel: expand1x1_planes + expand3x3_planes
  """

  def __init__(self, inplanes, squeeze_planes, expand1x1_planes,
     expand3x3_planes):
    super(Fire, self).__init__()
```

```
    self.inplanes = inplanes
    self.activation = nn.ReLU(inplace=True)
    self.squeeze = nn.Conv2d(inplanes, squeeze_planes, kernel_size=1)
    self.expand1x1 = nn.Conv2d(squeeze_planes, expand1x1_planes,
       kernel_size=1)
    self.expand3x3 = nn.Conv2d(squeeze_planes, expand3x3_planes,
       kernel_size=3, padding=1)

  def forward(self, x):
    x = self.activation(self.squeeze(x))
    return torch.cat([self.activation(self.expand1x1(x)),
       self.activation(self.expand3x3(x))], 1)

class MedianPool2d(nn.Module):
  """ Median pool (usable as median filter when stride=1) module.
  reference:
     https://gist.github.com/rwightman/f2d3849281624be7c0f11c85c87c1598
  Args:
    kernel_size: size of pooling kernel, int or 2-tuple
    stride: pool stride, int or 2-tuple
    padding: pool padding, int or 4-tuple (l, r, t, b) as in pytorch F.pad
    same: override padding and enforce same padding, boolean
  """
  def __init__(self, kernel_size=3, stride=1, padding=0, same=True):
    super(MedianPool2d, self).__init__()
    self.k = _pair(kernel_size)
    self.stride = _pair(stride)
    self.padding = _quadruple(padding) self.same = same

  def _padding(self, x):
    if self.same:
      ih, iw = x.size()[2:]
      if ih % self.stride[0] == 0:
        ph = max(self.k[0] - self.stride[0], 0)
      else:
        ph = max(self.k[0] - (ih % self.stride[0]), 0)
      if iw % self.stride[1] == 0:
        pw = max(self.k[1] - self.stride[1], 0)
      else:
        pw = max(self.k[1] - (iw % self.stride[1]), 0)
      pl = pw // 2
      pr = pw - pl
      pt = ph // 2
      pb = ph - pt
      padding = (pl, pr, pt, pb)
    else:
      padding = self.padding
```

```
    return padding

  def forward(self, x):
          x = F.pad(x, self._padding(x), mode='reflect')
    x = x.unfold(2, self.k[0], self.stride[0]).unfold(3, self.k[1],
       self.stride[1])
    x = x.contiguous().view(x.size()[:4] + (-1,)).median(dim=-1)[0]
    return x


class Backbone(nn.Module):
  """
    Class for Squeezeseg. Subclasses PyTorch's own "nn" module
  """

  def __init__(self, params):
      super(Backbone, self).__init__()
    print("Using Yau Backbone")

    self.use_range = params["input_depth"]["range"]
    self.use_xyz = params["input_depth"]["xyz"]
    self.use_remission = params["input_depth"]["remission"]
    self.drop_prob = params["dropout"]
    self.OS = params["OS"]
    self.mobilenet = torch.hub.load('pytorch/vision:v0.6.0',
       'mobilenet_v2', pretrained=True).cuda()
    self.mobilenet.eval()

       self.input_depth = 0
    self.input_idxs = []
    if self.use_range:
      self.input_depth += 1
      self.input_idxs.append(0)
    if self.use_xyz:
      self.input_depth += 3
      self.input_idxs.extend([1, 2, 3])
    if self.use_remission:
      self.input_depth += 1
      self.input_idxs.append(4)
        self.input_depth += 1 # proj_mask
    self.input_idxs.append(5)

    print("Depth of backbone input = ", self.input_depth)


    self.strides = [2, 2, 2, 2]
    # check current stride
```

```
current_os = 1
for s in self.strides:
  current_os *= s
print("Original OS: ", current_os)

# make the new stride
if self.OS > current_os:
  print("Can't do OS, ", self.OS,
       " because it is bigger than original ", current_os)
else:

  for i, stride in enumerate(reversed(self.strides), 0):
    if int(current_os) != self.OS:
      if stride == 2:
        current_os /= 2
        self.strides[-1 - i] = 1
    if int(current_os) == self.OS:
      break
  print("New OS: ", int(current_os))
  print("Strides: ", self.strides)

# encoder
self.conv1a = nn.Sequential(nn.Conv2d(self.input_depth, 64,
   kernel_size=3, stride=[1, self.strides[0]], padding=1),
   nn.ReLU(inplace=True))
self.conv1b = nn.Conv2d(self.input_depth, 64, kernel_size=1, stride=1,
   padding=0)
self.maxpool1 = nn.MaxPool2d(kernel_size=3, stride=[1,
   self.strides[1]], padding=1)
self.fire2 = Fire(64 + 32, 16, 64, 64) # fuse 7th
self.fire3 = Fire(128, 16, 64, 64)
self.maxpool2 = nn.MaxPool2d(kernel_size=3, stride=[1,
   self.strides[2]], padding=1)
self.fire4 = Fire(128 + 96, 32, 128, 128) # fuse 14th
self.fire5 = Fire(256, 32, 128, 128)
self.maxpool3 = nn.MaxPool2d(kernel_size=3, stride=[1,
   self.strides[3]], padding=1)
self.fire6 = Fire(256, 48, 192, 192)
self.fire7 = Fire(384 + 1280, 48, 192, 192) # fuse 19th
self.fire8 = Fire(384, 64, 256, 256)
self.fire9 = Fire(512, 64, 256, 256)
self.medpool3 = MedianPool2d()
self.medpool5 = MedianPool2d(kernel_size=5)
self.medpool7 = MedianPool2d(kernel_size=7)
self.medpool13 = MedianPool2d(kernel_size=13)
self.medpool29 = MedianPool2d(kernel_size=29)
```

```
  # output
  self.dropout = nn.Dropout2d(self.drop_prob)

  self.last_channels = 512

def run_layer(self, x, layer, skips, os):
    y = layer(x)
    if y.shape[2] < x.shape[2] or y.shape[3] < x.shape[3]:
      skips[os] = x.detach()
      os *= 2
    x = y
    return x, skips, os

def run_mobilenet(self, x):
  rgb_features = {}
  for i, layer in enumerate(self.mobilenet.features):
    x = layer(x)
    if i == 6:
      rgb_features['7th'] = x.clone().detach()
    elif i == 13:
      rgb_features['14th'] = x.clone().detach()
    elif i == 18:
      rgb_features['19th'] = x.clone().detach()
  return rgb_features

def fill_missing_points(self, tensor, mask):
  """
  Fill missing points in `tensor` indicated by `mask`
  Args:
     tensor: any H * W tensor
     mask: boolean mask where `False` indicates missing points
  Returns:
     median: filled tensor
  """
  eps = 1e-6
  H, W = tensor.shape[0], tensor.shape[1]
  assert H % 2 == 0
  device = tensor.device
  tensor = tensor * mask
  # repeatedly apply median filter
  median = tensor.clone()
  medpools = [self.medpool3, self.medpool5, self.medpool7,
     self.medpool13, self.medpool29]
  for medpool in medpools:
     median = median +
        medpool(median.unsqueeze(0).unsqueeze(0)).squeeze() *
        torch.logical_not(mask)
```

```
        mask = median.abs() > eps
    return median

def get_rgb_feature(self, range_img, rgb_features, calib_matrix):
    """get ready-to-fuse rgb features
    Note: Now only support batch size == 1!
    Args:
    range_img: batchsize=1 * ch * H * W tensor, channel = [r, x, y, z,
        remission, proj_mask]
    rgb_features: dict[rgb_layer_name] of rgb features, which are ch * H *
        W tensors
    Returns:
    corresponding_features: dict[sqseg_layer_name] of processed features
    """
    xyz = {}
    names = ['fire2', 'fire4', 'fire7']
    names_rgb = ['7th', '14th', '19th']
    strides = {'fire2':4, 'fire4':8, 'fire7':16}
    device = range_img.device
    assert range_img.shape[0] == 1, "batch size != 1, but now only support
        batch size == 1"
    range_img = range_img[0]
    a = range_img[1:4, :, ::2].clone().detach() # x, y, z
    mask = range_img[4, :, ::2].clone().detach().bool()
    for name in names:
      a, mask = a[:, :, ::2], mask[:, ::2] # only downsample on width
      li = []
      for i in range(3): # xyz
        li.append(self.fill_missing_points(a[i].clone(), mask)) # with
            missing points now
      xyz[name] = torch.stack(li, dim=0).reshape(3, -1) # flatten

    # lidar_points -> RGB
    rgb_idx = {}
    for layer in names:
      # corresponding row and column on RGB
      rgb_idx[layer] = torch.mm(calib_matrix, torch.cat((xyz[layer],
          torch.ones((1, xyz[layer].shape[1]), device=device))))
      rgb_idx[layer][:2, :] /= rgb_idx[layer][2, :] # normalize
      rgb_idx[layer] = rgb_idx[layer][:2, :] # discard useless channel
      rgb_idx[layer] = rgb_idx[layer].reshape(2, range_img.shape[1],
          range_img.shape[2] // strides[layer]) # reshape to the same size as
          range feature


    # clamp the out-of-bound points inside
    for na, nb in zip(names, names_rgb):
```

```
        H, W = rgb_features[nb].shape[2], rgb_features[nb].shape[3]

        rgb_idx[na][0, :, :] = rgb_idx[na][0, :, :].clamp(min=0, max=H - 1)
        rgb_idx[na][1, :, :] = rgb_idx[na][1, :, :].clamp(min=0, max=W - 1)

    # retrieve RGB feature by correspondence (flow)

    flow = {}
    flow['fire2'] = torch.round(rgb_idx['fire2'] / 8).long() # mobilenetv2
        7th
    flow['fire4'] = torch.round(rgb_idx['fire4'] / 16).long() # mobilenetv2
        14th
    flow['fire7'] = torch.round(rgb_idx['fire7'] / 32).long() # mobilenetv2
        19th

    corresponding_features = {}

    corresponding_features['fire2'] = rgb_features['7th'][0, :,
        flow['fire2'][0, :], flow['fire2'][1, :]]
    corresponding_features['fire4'] = rgb_features['14th'][0, :,
        flow['fire4'][0, :], flow['fire4'][1, :]]
    corresponding_features['fire7'] = rgb_features['19th'][0, :,
        flow['fire7'][0, :], flow['fire7'][1, :]]

    return corresponding_features

def forward(self, x, rgb_image, calib_matrix):
    # fuse preparation
    rgb_features = self.run_mobilenet(rgb_image)
    assert len(calib_matrix) == 1, "now only support batch size == 1"
    calib_matrix = calib_matrix[0]
    features = self.get_rgb_feature(x, rgb_features, calib_matrix) #
        features ready to concat

    # filter input
    x = x[:, self.input_idxs]

    # run cnn
    # store for skip connections
    skips = {}
    os = 1

    # encoder
    skip_in = self.conv1b(x)
    x = self.conv1a(x)

    skips[1] = skip_in.detach()
```

```
    os *= 2

    x, skips, os = self.run_layer(x, self.maxpool1, skips, os)

    x = torch.cat([x, features['fire2'].unsqueeze(0)], dim=1)
    x, skips, os = self.run_layer(x, self.fire2, skips, os)
    x, skips, os = self.run_layer(x, self.fire3, skips, os)
    x, skips, os = self.run_layer(x, self.dropout, skips, os)
    x, skips, os = self.run_layer(x, self.maxpool2, skips, os)
    x = torch.cat([x, features['fire4'].unsqueeze(0)], dim=1)
    x, skips, os = self.run_layer(x, self.fire4, skips, os)
    x, skips, os = self.run_layer(x, self.fire5, skips, os)
    x, skips, os = self.run_layer(x, self.dropout, skips, os)
    x, skips, os = self.run_layer(x, self.maxpool3, skips, os)
    x, skips, os = self.run_layer(x, self.fire6, skips, os)
    x = torch.cat([x, features['fire7'].unsqueeze(0)], dim=1)
    x, skips, os = self.run_layer(x, self.fire7, skips, os)
    x, skips, os = self.run_layer(x, self.fire8, skips, os)
    x, skips, os = self.run_layer(x, self.fire9, skips, os)
    x, skips, os = self.run_layer(x, self.dropout, skips, os)

    return x, skips

  def get_last_depth(self):
    return self.last_channels

  def get_input_depth(self):
    return self.input_depth
```

B.2 Data Pipeline

```
class SemanticKitti(Dataset):

  def __init__(self, root,
               sequences,
               labels,
               color_map,
               learning_map,
               learning_map_inv,
               sensor,
               max_points=150000,
               gt=True):

    self.root = os.path.join(root, "sequences")
    self.sequences = sequences
```

```
self.labels = labels
self.color_map = color_map
self.learning_map = learning_map
self.learning_map_inv = learning_map_inv
self.sensor = sensor
self.sensor_img_H = sensor["img_prop"]["height"]
self.sensor_img_W = sensor["img_prop"]["width"]
self.sensor_img_means = torch.tensor(sensor["img_means"],
                            dtype=torch.float)
self.sensor_img_stds = torch.tensor(sensor["img_stds"],
                           dtype=torch.float)
self.sensor_fov_up = sensor["fov_up"]
self.sensor_fov_down = sensor["fov_down"]
self.max_points = max_points
self.gt = gt

self.nclasses = len(self.learning_map_inv)

if os.path.isdir(self.root):
  print("Sequences folder exists! Using sequences from %s" % self.root)
else:
  raise ValueError("Sequences folder doesn't exist! Exiting...")

assert(isinstance(self.labels, dict))
assert(isinstance(self.color_map, dict))
assert(isinstance(self.learning_map, dict))
assert(isinstance(self.sequences, list))

self.scan_files = []
self.label_files = []
self.rgb_files = []

# fill in with names, checking that all sequences are complete
for seq in self.sequences:

  seq = '{0:02d}'.format(int(seq))

  print("parsing seq {}".format(seq))

  scan_path = os.path.join(self.root, seq, "velodyne")
  label_path = os.path.join(self.root, seq, "labels")
  rgb_path = os.path.join(self.root, seq, "image_2")

  scan_files = [os.path.join(dp, f) for dp, dn, fn in os.walk(
     os.path.expanduser(scan_path)) for f in fn if is_scan(f)]
  label_files = [os.path.join(dp, f) for dp, dn, fn in os.walk(
     os.path.expanduser(label_path)) for f in fn if is_label(f)]
```

```
    rgb_files = [os.path.join(dp, f) for dp, dn, fn in os.walk(
       os.path.expanduser(rgb_path)) for f in fn if is_rgb(f)]

    if self.gt:
      assert(len(scan_files) == len(label_files))

    self.scan_files.extend(scan_files)
    self.label_files.extend(label_files)
    self.rgb_files.extend(rgb_files)

  # sort for correspondance
  self.scan_files.sort()
  self.label_files.sort()
  self.rgb_files.sort()

  print("Using {} scans from sequences {}".format(len(self.scan_files),
                                                  self.sequences))

def __getitem__(self, index):
     scan_file = self.scan_files[index]
  rgb_file = self.rgb_files[index]
  if self.gt:
    label_file = self.label_files[index]

     if self.gt:
    scan = SemLaserScan(self.color_map,
                    project=True,
                    H=self.sensor_img_H,
                    W=self.sensor_img_W,
                    fov_up=self.sensor_fov_up,
                    fov_down=self.sensor_fov_down)
  else:
    scan = LaserScan(project=True,
                 H=self.sensor_img_H,
                 W=self.sensor_img_W,
                 fov_up=self.sensor_fov_up,
                 fov_down=self.sensor_fov_down)

  # open and obtain scan
  scan.open_scan(scan_file)
  if self.gt:
    scan.open_label(label_file)
    # map unused classes to used classes (also for projection)
    scan.sem_label = self.map(scan.sem_label, self.learning_map)
    scan.proj_sem_label = self.map(scan.proj_sem_label, self.learning_map)

  # make a tensor of the uncompressed data (with the max num points)
```

```
unproj_n_points = scan.points.shape[0]
unproj_xyz = torch.full((self.max_points, 3), -1.0, dtype=torch.float)
unproj_xyz[:unproj_n_points] = torch.from_numpy(scan.points)
unproj_range = torch.full([self.max_points], -1.0, dtype=torch.float)
unproj_range[:unproj_n_points] = torch.from_numpy(scan.unproj_range)
unproj_remissions = torch.full([self.max_points], -1.0,
   dtype=torch.float)
unproj_remissions[:unproj_n_points] = torch.from_numpy(scan.remissions)
if self.gt:
  unproj_labels = torch.full([self.max_points], -1.0, dtype=torch.int32)
  unproj_labels[:unproj_n_points] = torch.from_numpy(scan.sem_label)
else:
  unproj_labels = []

# get points and labels
proj_range = torch.from_numpy(scan.proj_range).clone()
proj_xyz = torch.from_numpy(scan.proj_xyz).clone()
proj_remission = torch.from_numpy(scan.proj_remission).clone()
proj_mask = torch.from_numpy(scan.proj_mask)
if self.gt:
  proj_labels = torch.from_numpy(scan.proj_sem_label).clone()
  proj_labels = proj_labels * proj_mask
else:
  proj_labels = []
proj_x = torch.full([self.max_points], -1, dtype=torch.long)
proj_x[:unproj_n_points] = torch.from_numpy(scan.proj_x)
proj_y = torch.full([self.max_points], -1, dtype=torch.long)
proj_y[:unproj_n_points] = torch.from_numpy(scan.proj_y)
proj = torch.cat([proj_range.unsqueeze(0).clone(),
              proj_xyz.clone().permute(2, 0, 1),
              proj_remission.unsqueeze(0).clone()])
proj = (proj - self.sensor_img_means[:, None, None]
      ) / self.sensor_img_stds[:, None, None]
proj = proj * proj_mask.float() # missing pixel set to all 0s
proj = torch.cat([proj, proj_mask.unsqueeze(0).clone().float()]) # add
   channel on top of sqseg

# get name and sequence
path_norm = os.path.normpath(scan_file)
path_split = path_norm.split(os.sep)
path_seq = path_split[-3]
path_name = path_split[-1].replace(".bin", ".label")

# crop to front 90 deg
proj = proj[:, :, 768:1280]
proj_labels = proj_labels[:, 768:1280]
```

```
  # get rgb image
  raw_image = Image.open(rgb_file)
  preprocess = transforms.Compose([
     transforms.ToTensor(),
     transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
        0.225])
  ])
  rgb_image = preprocess(raw_image)

  calib_path = os.path.join(self.root, path_seq, "calib.txt")
  # read calib file
  calib = {}
  with open(calib_path, 'r') as file:
    for line in file.readlines():
      key, value = line.split(":", 1)
      value = torch.tensor([float(i) for i in value.split()],
         dtype=torch.float).reshape(3, 4)
      calib[key] = value
  Tr = torch.cat((calib['Tr'], torch.tensor([[0., 0., 0., 1.]])))
  P2 = calib['P2']
  calib_matrix = torch.mm(P2, Tr)

  return proj, proj_labels, rgb_image, calib_matrix

def __len__(self):
  return len(self.scan_files)

@staticmethod
def map(label, mapdict):
  maxkey = 0
  for key, data in mapdict.items():
    if isinstance(data, list):
      nel = len(data)
    else:
      nel = 1
    if key > maxkey:
      maxkey = key
  if nel > 1:
    lut = np.zeros((maxkey + 100, nel), dtype=np.int32)
  else:
    lut = np.zeros((maxkey + 100), dtype=np.int32)
  for key, data in mapdict.items():
    try:
      lut[key] = data
    except IndexError:
      print("Wrong key ", key)
  return lut[label]
```