参赛队员姓名：吴宇伦

中学：上海市建平中学

省份：上海

国家/地区：中国

指导教师姓名：许娉婷

论文题目：DenseFuseNet: 3D Semantic Segmentation in the Context of Autonomous Driving with Dense Correspondence

本参赛团队声明所提交的论文是在指导老师指导下进行的研究工作和取得的研究成果。尽本团队所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果。若有不实之处，本人愿意承担一切相关责任。

参赛队员：吴宇伦　　指导老师：许娉婷

2020 年　9 月　14　日

DenseFuseNet: 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 the prevailing automatic driving system is to study how to integrate the multi-modal perception subsystem 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 famous 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[7]) has sufficiently proved that merging the 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 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 4-step pipeline to establish a connection between lidar and RGB camera models, matching and fusing the features of the lidar and RGB models. We also defined two kinds of inaccuracies (missing pixels and covered points) in point cloud projection and did a numerical analysis on them. Furthermore, we proposed a 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[17] we reproduced. All code is made open-source on https://github.com/ID10T/DenseFuseNet.

Keywords: *DenseFuseNet, autonomous driving, 3D semantic segmentation, dense correspondence, sensor fusion*

Contents

1. Introduction

Autonomous vehicles are becoming increasingly popular among consumers and venture capitalists. In the first half of 2020, giants and startups in autonomous driving were financed with more than 7.6 billion USD[1]. In the foreseeable future, autonomous vehicles will eliminate traditional cars and reform our society. Researchers and business leaders are optimistic that autonomous driving will bring valuable benefits to human society by massively boosting productivity. More importantly, autonomous driving technology is expected to solve a critical cause of mortality: traffic accidents[2].

A reliable autonomous driving system must have a robust perception system. A perception system usually constitutes a suite of sensors to gather information on the environment and an AI model to recognize objects in its surroundings. This paper focuses on an important task in perception: Open Space 3D Semantic Segmentation.

In this section, we first briefly introduce open space 3D semantic segmentation, one of the central tasks of autonomous driving. Then we introduce the idea of sensor fusion, a technique to improve the segmentation performance. Finally, we propose DenseFuseNet, a sensor fusion model with a general 4-step pipeline to fuse arbitrary space-invariant lidar and RGB camera models.

1.1. Open Space 3D Semantic Segmentation

As there are requirements in autonomous driving, the main element of the perception system is to recognize objects in the surrounding traffic conditions, which is usually treated as open space 3D semantic segmentation in the traffic scene. Specifically, it requires perception systems to accurately detect the position, shape, and type of traffic lights, cars, and pedestrians in 3D space. An example of open space semantic segmentation is in Fig. 1. The precise definition of open space semantic segmentation and why we choose this task are covered in Section 2.3.

There are two reasons why open space 3D semantic segmentation is so important in autonomous driving. 1. object classification is necessary since the autonomous driving system should respond differently to different objects. 2. A point-wise segmentation is also required because it is essential for autonomous vehicles to know the exact shape of the objects while maneuvering in the road.

For better segmentation qualities, we make use of a technique called sensor fusion.

1.2. Sensor Fusion

There are two major sensors of the perception system, lidars and RGB cameras; both can be used to enhance open space 3D semantic segmentation. The detailed comparison between lidars and RGB cameras are in Section 3.2. Since lidars and cameras produce data in different formats, it is a thought-provoking and unsolved problem how to design a general model that takes both RGB images and point clouds as inputs and outputs the classified point cloud according to the category of the object (bicycle, person, etc.).

A recent work, FuseSeg[7], proposes a way to fuse features between a lidar model and an RGB image model. It uses farthest point sampling and first-order spline interpolation to guide the fusion between a lidar model and an RGB image model, improving accuracy on 3D semantic segmentation. However, it does have several drawbacks: 1. Not scalable computation time (high time complexity). 2. Distortion due to interpolation (Figure. 3). We will be elaborate on these problems later. 3. It is close-sourced,

[1] finance.eastmoney.com

[2] According to WHO, 1.35 million lives per year are gone due to traffic accidents

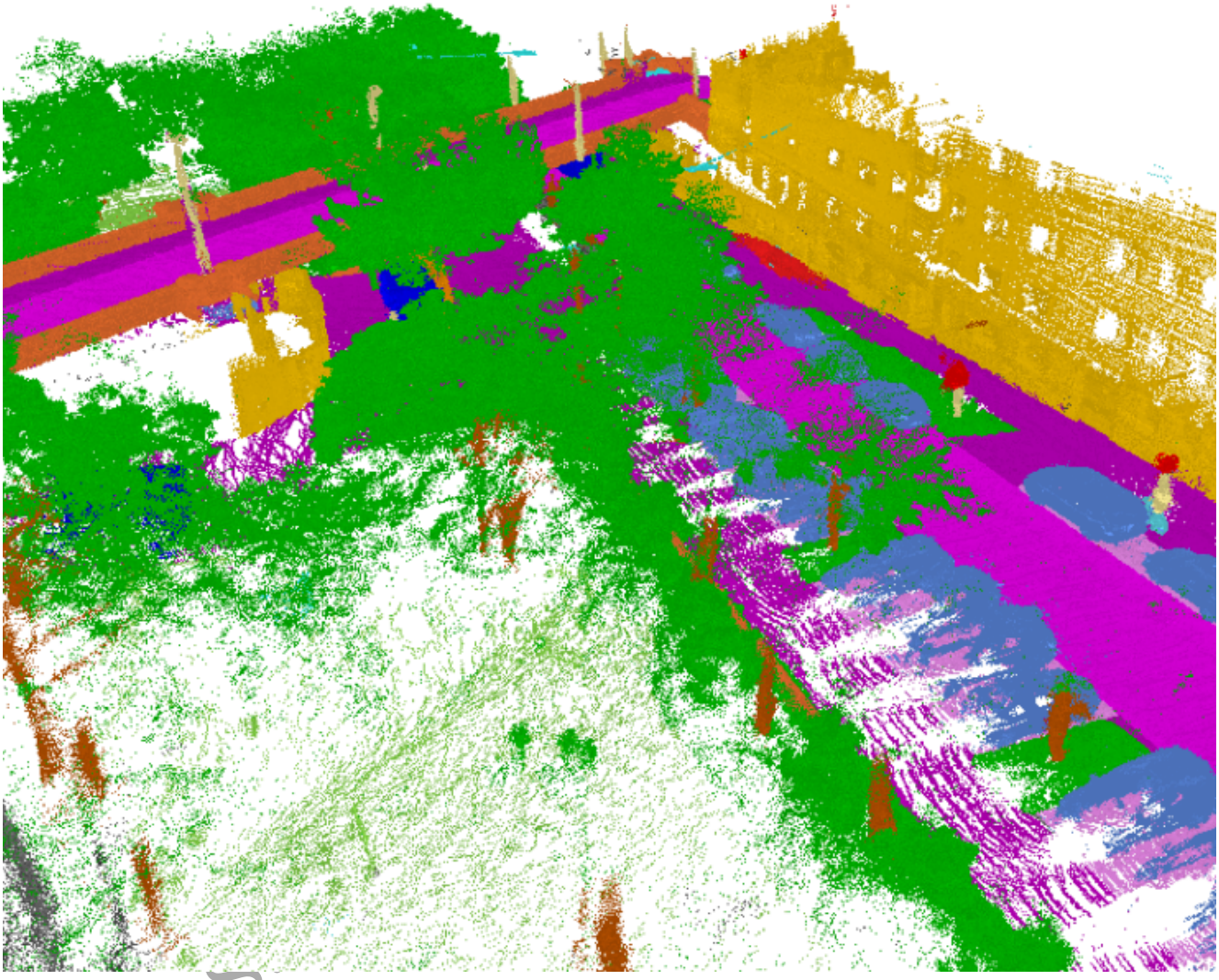


Figure 1. An result of open space 3D semantic segmentation from [2].

which makes reproducing practically impossible. To solve these problems, we propose a new approach called DenseFuseNet.

1.3. Proposed Method: DenseFuseNet

We propose a novel method, named DenseFuseNet, for 3D semantic segmentation. In our model, we exploit the calibration data between lidar and cameras to establish a dense and accurate point correspondence between arbitrary layers of a lidar model (SqueezeSeg[17]) and a pretrained RGB image model (MobileNetv2[15]), which we call U-shaped correspondence. We propose a 4-step pipeline to establish

the U-shaped correspondence. Then, we warp and fuse the features of the pretrained MobileNetv2 into SqueezeSeg according to the U-shaped correspondence.

Our 4-step pipeline has a more scalable run time and generates more accurate correspondence, which results in better fusion quality. Moreover, it can be generalized to any model that is space invariant. As we previously mentioned, DenseFuseNet is based on SqueezeSeg, which is not close to the best model for open space 3D semantic segmentation, but the observed improvement on SqueezeSeg may also materialize in other baseline models. In other words, we can apply our method to the state-of-the-art lidar models and RGB models for even better performance.

We tested DenseFuseNet on SemanticKITTI[2], a 3D semantic segmentation dataset containing 23,201 lidar scans and as many RGB images. DenseFuseNet achieves a noticeable improvement of 5.8% and 14.2% in mIoU and accuracy respectively, well beyond the plain SqueezeSeg we reproduced.

Point cloud projection is a common preprocess technique to embed 3D point clouds into 2D images for easier processing. However, in practice, the inaccuracy of the lidar sensor could affect projection quality. In this paper, we defined and did a numerical analysis on two kinds of inaccuracies, which we name missing pixels and covered points. We also designed an efficient, GPU-friendly algorithm to resolve the problem of missing pixels.

Our method is applicable to existing models and empowers autonomous vehicles' perception systems for recognizing road objects. Autonomous driving companies can fine-tune and adapt this method to better combine and utilize the power of lidar and camera. An increasing number of new vehicles embodies the feature of autonomous driving. As autonomous driving systems gradually replace human drivers, people have good wishes for them to protect lives from traffic accidents. We hope our work will help to bring convenience and safety to humanity when people hop into their cars and hit the road.

2. Related Work

In this section, we review some closely related tasks and techniques for semantic segmentation and instance segmentation on RGB images and point clouds, respectively. Besides, we mention some representative works related to point cloud projection and sensor fusion.

2.1. Semantic Segmentation on RGB Images

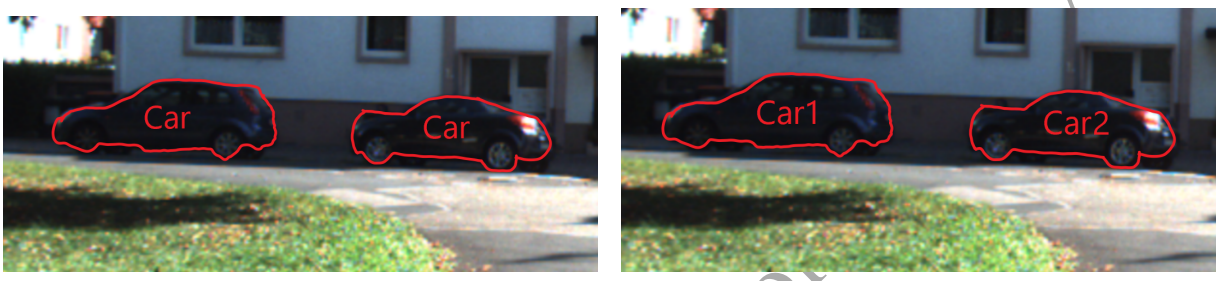
Semantic segmentation is the task of clustering parts of an image together, which belong to the same object class, and assign a label to the image segment. These labels could include cars, pedestrians, cyclists, etc. A sample of semantic segmentation on RGB images is shown in Fig. 2(a). One of the earliest and most innovative works on semantic segmentation[3] is FCN[8]. It used a transposed convolutional layers to upsample extracted image features and restore pixel category from abstract features. Another work U-Net[14] laid the foundation of encoder-decoder architecture of semantic segmentation models. It establishes pathways to crop and concatenate early feature maps with the feature maps after upsampling, enabling the information and gradient to better propagate through the entire model. The state-of-the-art of semantic segmentation to the time is HRNet[16], which maintains a high-resolution representation throughout the network, and does parallel information exchange between multiple resolution representations.

[3]In this section, we simply use "semantic segmentation" to stand for semantic segmentation on RGB images for convenience.

2.2. Instance Segmentation on RGB Images

Semantic segmentation means to segment objects by their classes (car, pedestrian. . .), while instance segmentation takes a step further to separate different instances (car0, car1, pedestrian0, pedestrian4. . .). A visual comparison of semantic segmentation and instance segmentation is in Fig. 2. One of the most representative work in instance segmentation is Mask R-CNN[6], which combines a object detection model, Faster-RCNN[13], and FCN[8]. Mask R-CNN used a region proposal network to locate "interesting" regions 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.



(a) Semantic segmentation (b) Instance segmentation

Figure 2. A comparison between semantic segmentation and instance segmentation. Instance segmentation distinguishes different instances of the same class, but semantic segmentation does not.

2.3. 3D Semantic Segmentation on Point Clouds

3D semantic segmentation is analogous to semantic segmentation on 2D images. 2D semantic segmentation means to assign each pixel a label by their class. Similarly, 3D semantic segmentation means to assign a label to every point in a point cloud (an unordered set of 3D points, usually provided by a lidar). One step further, 3D semantic segmentation can be conducted on closed scenes (a room) or open space (on a highway). In the "closed" case, the point clouds to be segmented are usually small, static, and simple; thus, it has a looser demand on speed. Works like PointNet[12] treated a point cloud in its crude form, an unordered array of points. It uses a series of multi-layer perceptron to process the point array and uses a global max pooling to retrieve a feature that is invariant to the input permutation. However, it only extracts global features, which are inconsistent with the spirit of CNN (extracting local features per layer). Moreover, it is incredibly slow when the input size is large. Therefore, it doesn't quite fit the scenario of autonomous driving.

In the context of autonomous driving, lidar point clouds produced by the lidar on cars are massive and complex, and the quality of point clouds can vary significantly in different environments. Semantic segmentation on this kind of point clouds is known as "open space" 3D semantic segmentation.

Similarly, 3D instance segmentation distinguishes different objects of the same class on top of 3D semantic segmentation. While 3D instance segmentation might seem a more suitable task for autonomous driving since it provides more information for an autonomous driving system, it is at the same time a more demanding task, which usually requires heavy models that cannot perform inference in real-time. Currently, many works on autonomous driving consider semantic segmentation a vital task on autonomous driving, and many datasets like SemanticKITTI[2] only provide semantic labels rather than

instance labels. Moreover, semantic segmentation can be converted to instance segmentation, for example, by clustering methods [11]. Therefore, we could say that semantic segmentation is worthy of being studied for autonomous driving.

Open space 3D semantic segmentation is the main focus of this paper. In the following sections, we introduce some influential works on open 3D semantic segmentation.

2.4. 3D Semantic Segmentation with Point Cloud Projection

TThe idea of point cloud projection was first introduced in SqueezeSeg[17]. It uses a spherical projection to embed the 3D point cloud into a 2D grid. This embedding converts the previously sparse 3D point cloud into a dense 2D representation, calls a "lidar image". It permits the use of convolutional neural networks, which are faster and more parameter-efficient than other networks dealing with 3D point clouds directly. Following works like SqueezeSegv2[18] and SqueezeSegv3[19] further enhanced SqueezeSeg and achieved a performance boost so massive as they become the most accurate and efficient models in open space 3D semantic segmentation. SqueezeSegv2 introduced a Context Aggregation Module (CAM) to deal with the noise in the projected lidar image, and SqueezeSegv3 introduced the Spatially-Adaptive Convolution to apply different filters for different locations of the input lidar image. In other words, it considers the nature of lidar images that they change dramatically in different image regions. SqueezeSegv3 is one of the best-performing models in open space 3D semantic segmentation, but its inference speed drops to around 10 FPS. Aside from the SqueezeSeg series, many other works such as RangeNet++[9] also adopt point cloud projection.

In practice, due to the external causes like transparent material and precision error, inaccuracies can appear in the projected lidar image, which we categorize as "missing points" and "covered points". More on those in section 3.4.

The mechanics behind point cloud projection are covered in section 3.3.

Another recent work, 3D-MiniNet[1], also embeds point clouds to 2D representation. Different from point cloud projection, it uses a neural network to learn a 2D representation, then combining it with projected lidar images. 3D-MiniNet is also one of the top-performing models in open space 3D semantic segmentation.

2.5. Sensor Fusion in 3D Semantic Segmentation

One recent work, FuseSeg[7], improved the performance of SqueezeSeg[17] by fusing it with a pre-trained image classification model, MobileNetv2[15].

FuseSeg is an interesting work on sensor fusion. It fuses a SqueezeSeg and a MobileNetv2 together to improve segmentation accuracy. FuseSeg[7] performs FPS[4] on the lidar point cloud to select a set of "control points" on the lidar feature, calculating their corresponding positions on the RGB feature. It then uses first-order spline interpolation to establish a dense correspondence used to fuse the RGB feature to the lidar model. However, it has some imperfections, which we mentions in the next section.

3. Our method

FuseSeg's[7] approach has several drawbacks: 1. It can cause serious distortion (Fig.3), mainly due to interpolation on a small number of "control points", that is, only 48 "control points" are selected out of 32000+ total points per lidar scan. 2. Its method is hardly scalable since both FPS and first-order spline

[4]Farthest point sampling, which has a time complexity of $O(n^2)$.

Figure 3. The artifacts produced by FuseSeg[7]. There are noticeable ghosting such as around the cyclist.

interpolation have a time complexity of $O(n^2)$, which causes the inference time to increase dramatically (empirically proved in [7]) as the number of control points goes up. Though we can manually limit the number of control points, it is still problematic since we are going to have sensors with higher resolution in the future. 3. FuseSeg is close-sourced. That's why we can neither reproduce nor directly make comparisons.

In this section, we first introduce the sensors used in autonomous driving, as well as the point cloud format. On top of that, we first give a review on point cloud projection, a core preprocessing fashion to enable CNNs to process lidar point clouds. Then, we introduce the concept of missing pixels and control points, their impact on segmentation, and possible solutions. After that, we propose a 4-step approach to establish a relationship between the lidar model and the RGB model, and its implication on feature fusion. Finally, we covers the architecture of DenseFuseNet, which incorporates our filling algorithm and U-shaped correspondence.

3.1. Sensors

An autonomous vehicle is usually equipped with a variety of sensors, often a combination of one lidar and multiple cameras and radars[5]. Lidars and RGB cameras have pros and cons, respectively. Table 3.1 compares lidar with RGB cameras from several aspects.

As shown in the table, lidar sensors can provide accurate distance data in a decent range, but it can be obscured by rain and fog. On the other hand, cameras are cheap and can easily read signs and road marks, but they are "dumb" sensors that can only output planar images and lose sight at nighttime.

[5]Radars are mostly used as supplementary sensors, so they are out of the scope of this paper.

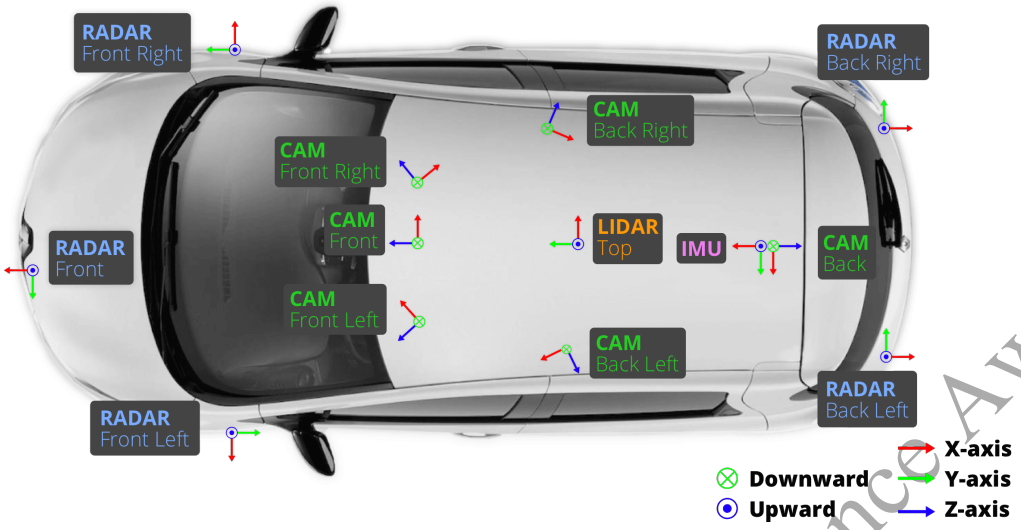


Figure 4. An standard sensor suite of an autonomous vehicle [3]

	RGB Camera	Lidar
Depth	Not directly available	Can be precisely retrieved
Environmental Dependence	Good illumination	Clean air (Fog and raindrops reflect laser)
Interference	None	Multiple lidars can interfere each other
Price	$50-1000	Very expensive, usually $70k+, but is dropping

Table 1. Comparison between a lidar and an RGB camera.

3.2. Lidars and Point Clouds

Lidars are arguably the most important sensor in autonomous driving. A lidar emits pulsed laser beams into its surroundings, and the waves bounce back from the objects and are received by the sensor. Then the lidar calculates the distance traveled of each laser beam with the time it took to return. It also calculates the reflectance of the surface the laser beam hits by measuring the intensity of the reflected beam.

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 points are denser close to the lidar since the lidar is the source of emitted laser beams.

With these pros and cons, lidars are widely adopted by various automotive corporations like Waymo, Lyft, Baidu, etc. as the primary sensor for their autonomous driving system. Lidar can be used in various ways to perceive surroundings, such as 3D object detection, point cloud segmentation, and SLAM. Lidar is useful in many fields like robotics, geography, and autonomous driving. Therefore, it's valuable to develop methods based on a lidar sensor.

3.3. Spherical Projection of Lidar Point Cloud

Many previous works on 3D semantic segmentation like [17] and [9] rely on spherical projection. Spherical projection is a way to embed a 3D point cloud into a 2D image, which is easier to process

using CNNs. It works well on lidar point clouds because, as stated in section 3.2, the lidar emits lasers in all directions. In other words, it is the center of the point cloud that if we let it be the origin and perform spherical projection on the point cloud, we should be able to retrieve an even grid of points in the 2D plane.

Formally, a 3D point cloud can be modeled as an unordered set of points $[x, y, z]$, and projected to a pixel $[u, v]$ in 2D space 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 arbitrary point in the 3D Lidar point cloud, and $[u, v]$ is the coordinate of the projected 2D pixel on the lidar image. h and w are the desired height and width of the output lidar image. $\text{fov}_{\text{up/down}}$ is the upper/lower field of view of the lidar sensor, and $\text{fov} = \text{fov}_{\text{down}} + \text{fov}_{\text{up}}$. The projected image is called the lidar image.

However, Spherical projection has imperfections. Due to the inherent inaccuracy of the lidar sensor, sometimes multiple lidar points are projected onto the same pixel of the lidar image, and some pixels end up with no points projected on it. The latter adds difficulties to feature fusion, and the former will make the average performance drawback 2.36% (in Section 3.1). Therefore, how to quantify and reduce their impact becomes an important problem in our research. To simplify the language, we introduce the concepts of missing pixels and covered points.

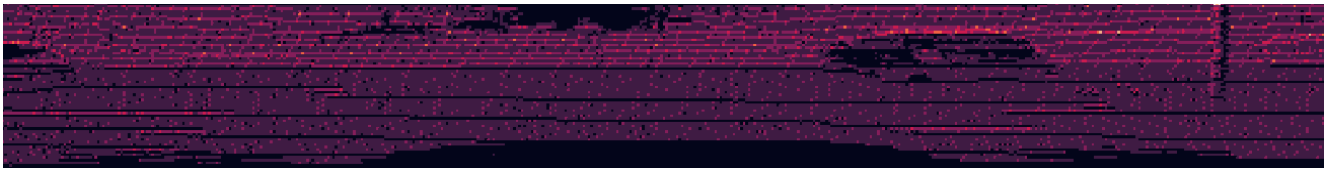
3.4. Missing pixels and covered points

In practice, spherical projection can introduce two main things that undermine semantic segmentation effectiveness: missing pixel and covered points. They are caused by the inherent inaccuracy of the lidar sensor and the environment. An example of missing pixels and covered points are in Fig.5.

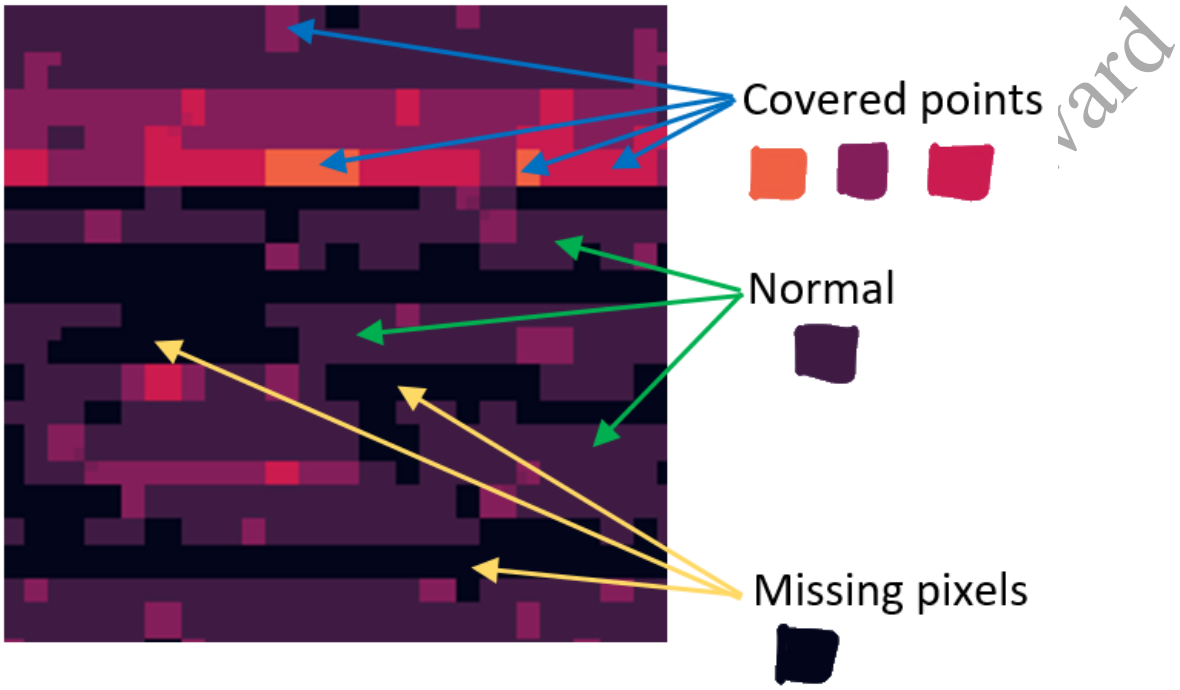
The missing points mean that there is no exact one-to-one projection relationship between the 3D lidar points and range images. In other words, it also implies that 1) many points in the 3D lidar points will project to the same position (pixel) in range images, and 2) many pixels in range images cannot trace back to their three-dimensional coordinates in 3D lidar points. Generally, missing pixels can be classified into three types: 1. the "pepper and salt" single-pixel missing caused by the inherent noise. 2. the stripe-shaped missing pixels caused by the structural inaccuracy (the inaccuracy in pitch angle) of the lidar sensor. 3. the large missing patch mainly caused by surfaces that don't reflect lasers like car windows and the sky.

Covered points emerge because sometimes multiple lidar points are projected into the same pixel, and some points are covered by others.

Covered points will introduce two problems: 1. the information that could improve the segmentation result is lost 2. the segmentation model 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. We assume the first problem wouldn't have too much impact according to [12]'s conclusion that a small subset of points is enough to segment an object. Problem 2 could be remedied by CRF or KNN post-processing, but it is out of scope in our paper. Still, we quantified its impact statistically in section 4.



(a) The frequency map



(b) Missing pixels and covered points

Figure 5. Missing pixels and covered points on the frequency map. The brightness of the frequency map describes how many lidar points are projected onto each 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.

3.5. Establishing the U-shaped correspondence

Our goal is to fuse the RGB camera model to the lidar model. Therefore, we attempted to find a U-shaped correspondence between the feature maps of two models. To explain the notion of U-shaped correspondence, we first introduce the concept of a dense correspondence. We formally define a dense correspondence f as:

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

Where **A**, **B** are rectangular feature maps, and (i, j) denotes image coordinates. In simple languages, a dense correspondence is a pixel to pixel mapping between two feature maps. Also, note that dense correspondence has transitivity.

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 has. Note that in order to perform a good 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 then the feature fusion would be meaningful to the CNN.

To establish a U-shaped correspondence, we propose a 4-step pipeline in this section. 1. Fill the missing pixels in the lidar image. 2. Establish a dense correspondence from lidar feature to lidar image. 3. Establish a dense correspondence from lidar image to RGB image. 4. Establish a dense correspondence from RGB image to RGB feature. Then, we can obtain a U-shaped correspondence by combining the correspondences in step (2)(3)(4). An overall view of the 4-step pipeline is visualized in Fig.6.

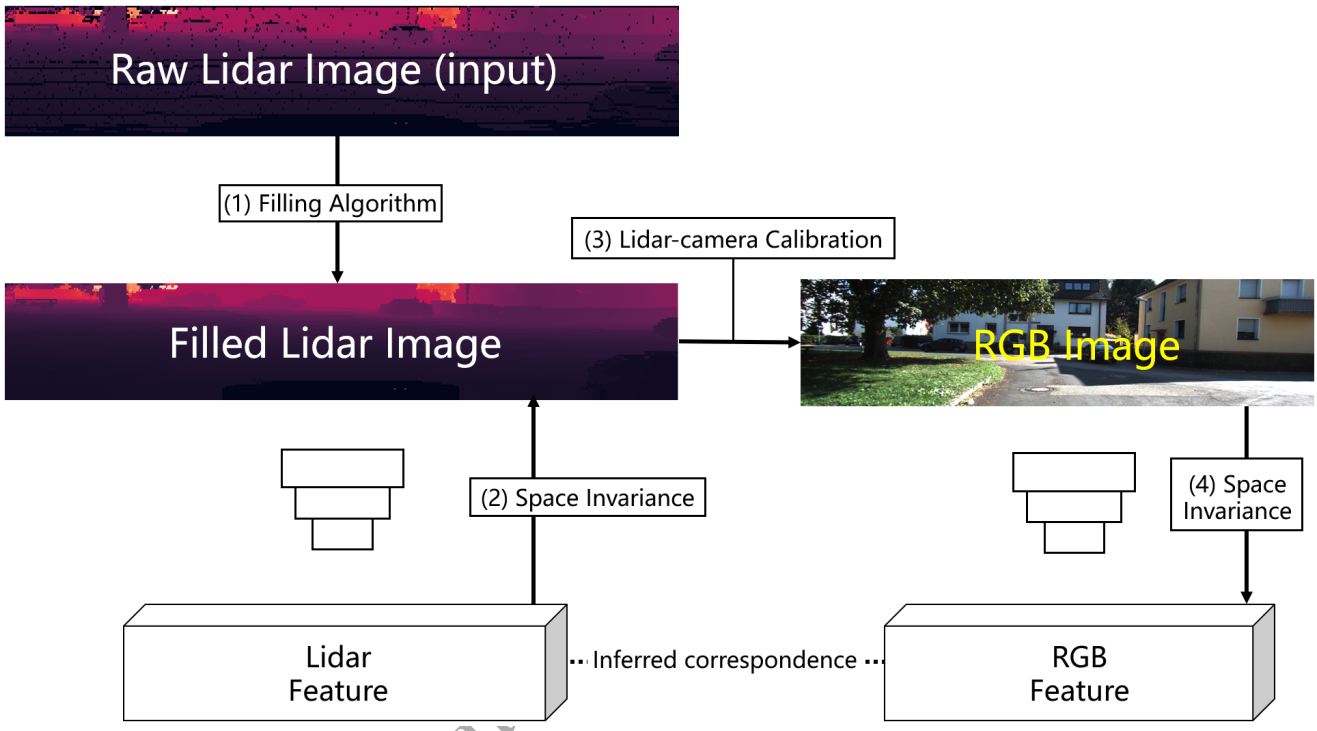


Figure 6. An overview of the 4-step pipeline to establish U-shaped correspondence. (1) fills the missing pixels in the input lidar image (2) and (4) denotes the dense correspondence established by space invariance. (3) is the dense correspondence established by lidar-camera calibration.

3.5.1 Filling Missing Pixels

To better establish dense correspondences, we preprocessed the lidar images by filling out missing pixels.

As in section 3.4, missing pixels are of three classes. We propose a unified method to deal with the "pepper and salt" and stripe-shaped missing pixels together. We first apply a series of median filters with increasing kernel size on the lidar image, while always keeping the original data. In other words, we define a mask indicating the pixels we need to fill, and keep the rest untouched. Formal algorithms is in 1

With our filling algorithm, we successfully reduced the missing pixels percentage from 24.323% to 6.274%

A Failed Approach to Fill Missing Pixels In fact, we first tried to use a masked median filter combined heuristics to fill the entire lidar image. That failed approach is presented in this section.

Algorithm 1: Filling missing pixels

Input: lidar image, initial mask
Output: filled lidar image
filters ← 3x3, 5x5, 7x7, 13x13, ...; `// median filters of different kernel sizes`
for filter in filters **do**
    median ← filter(lidar image);
    lidar image ← lidar image + median × mask;
    mask ← 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.

In the early stage of our research, we tried to fill missing pixels by: 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) [6].

However, this algorithm was hard to compute in parallel since the size of s is variable. Thus, we had to seek out new methods that can be represented as mass tensor computation on GPUs. What we finally

[6]For example, fill the top half of the range channel with the global maximum, and fill the lower half with the local minimum.

came up with was the filling method stated in section 7. Our new method resulted in a 20x speedup compared with this one.

3.5.2 Lidar Feature to Lidar Image

In this step, we find the dense correspondence from lidar feature to lidar image. In a CNN, a pixel on the feature map is a convolution sum over a square 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.

In this process, we use the geometric center of the receptive fields of the feature map as the representative coordinates for each pixel on the feature map of layer $l+1$, and map from layer $l+1$ to layer l repeatedly to retrieve the correspondence between arbitrary layers of the network.

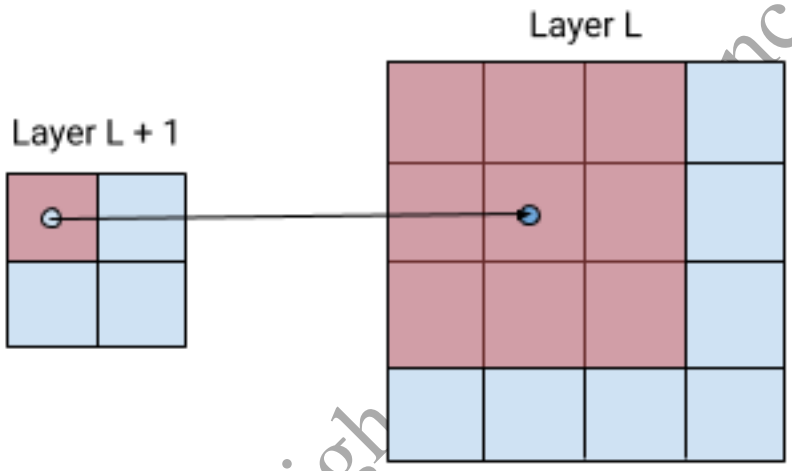


Figure 8. The correspondence between layer $l+1$ and layer l, with 3x3 kernel. Even kernel size also works by introducing floating number.

Then, we can map from layer $l+1$ to layer l repeatedly to retrieve a dense correspondence between two arbitrary layers of the network.

In this way, a dense correspondence is always available as long as we are able to determine the "center." Thus, our method is highly generalizable for most space-invariant models. For instance, one top-performing model of 3D semantic segmentation, 3D-MiniNet [1], once being the state-of-the-art on the Semantic-KITTI leaderboard, which is based on neighbor searches on grids, is also applicable by our method. In this case, we can turn 3D-MiniNet into a sensor fusion model in split seconds.

Specifically, to the model backbone we used in this paper, SqueezeSeg, all of its convolution layers is of kernel size 1x1 without padding, or 3x3 with a padding of 1 pixel, which means the center of the receptive fields of each pixel in the feature map of layer $l+1$ is at exactly the same coordinate in feature map l. Moreover, SqueezeSeg only downsamples on width. Therefore, we can induce a clean formula specific for the dense correspondence between layer a and b, where $a < b$:

$$A_b[i,j] \rightarrow A_a[i, j \times \text{downsample factor}] \tag{3}$$

3.5.3 Lidar Image to RGB Image

The best way to establish connections between lidar image and RGB image is to exploit the calibration file usually provided by every multi-modal dataset.

The calibration file specifies a 3x4 matrix **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:

$$\mathbf{x}_{\text{rgb}i} = \mathbf{P}_i \mathbf{Tr} \mathbf{x}_{\text{lidar}} \tag{4}$$

Where **Tr** is the transformation from the lidar coordinate to the 0-th camera coordinate. $\mathbf{P}_i$ is the transformation matrix from the 0-th camera coordinate to the i-th camera coordinate. Applying **Tr** then $\mathbf{P}_2$ translates points in the lidar coordinate $\mathbf{x}_{\text{lidar}} = [x, y, z, 1]^T$ to pixel positions on the second RGB camera $\mathbf{x}_{\text{rgb}} = [r, c, 1]^T$.

3.5.4 RGB Image to RGB Feature

Since the RGB model are also a CNN, we can use the method in Section 3.5.2 in reverse to establish the dense correspondence from RGB image to RGB feature:

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

3.5.5 Feature Fusion with U-shaped Correspondence

We combine three dense correspondences for a U-shaped correspondence f.

With the 4-step pipeline (filling + 3 dense correspondences), we are able to determine f between arbitrary lidar feature maps and arbitrary RGB feature maps. Then, we are granted the routes through which we can fuse features 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. There are more possible ways to fuse features but is out of the scope of this paper and might be covered in future works.

3.6. Model Architecture

We adopted the model structure of FuseSeg[7], which constitutes two models: SqueezeSeg[17] as segmentation backbone and MobileNetv2[15] as RGB feature extractor. We used the implementation of MobileNetv2 from Pytorch Hub[10], with weights pretrained on ImageNet[4]. We didn't bother fine-tuning the MobileNetv2 in this paper, even though it should improve performance.

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.

4. Experiments

In this section, we first introduce the SemanticKITTI, the dataset we use, and compare it with its predecessor, KITTI. Then, we did a quantitative analysis on missing pixels and covered points. Finally, we bring in the result of our model training and verified the effectiveness of the U-shaped correspondence.

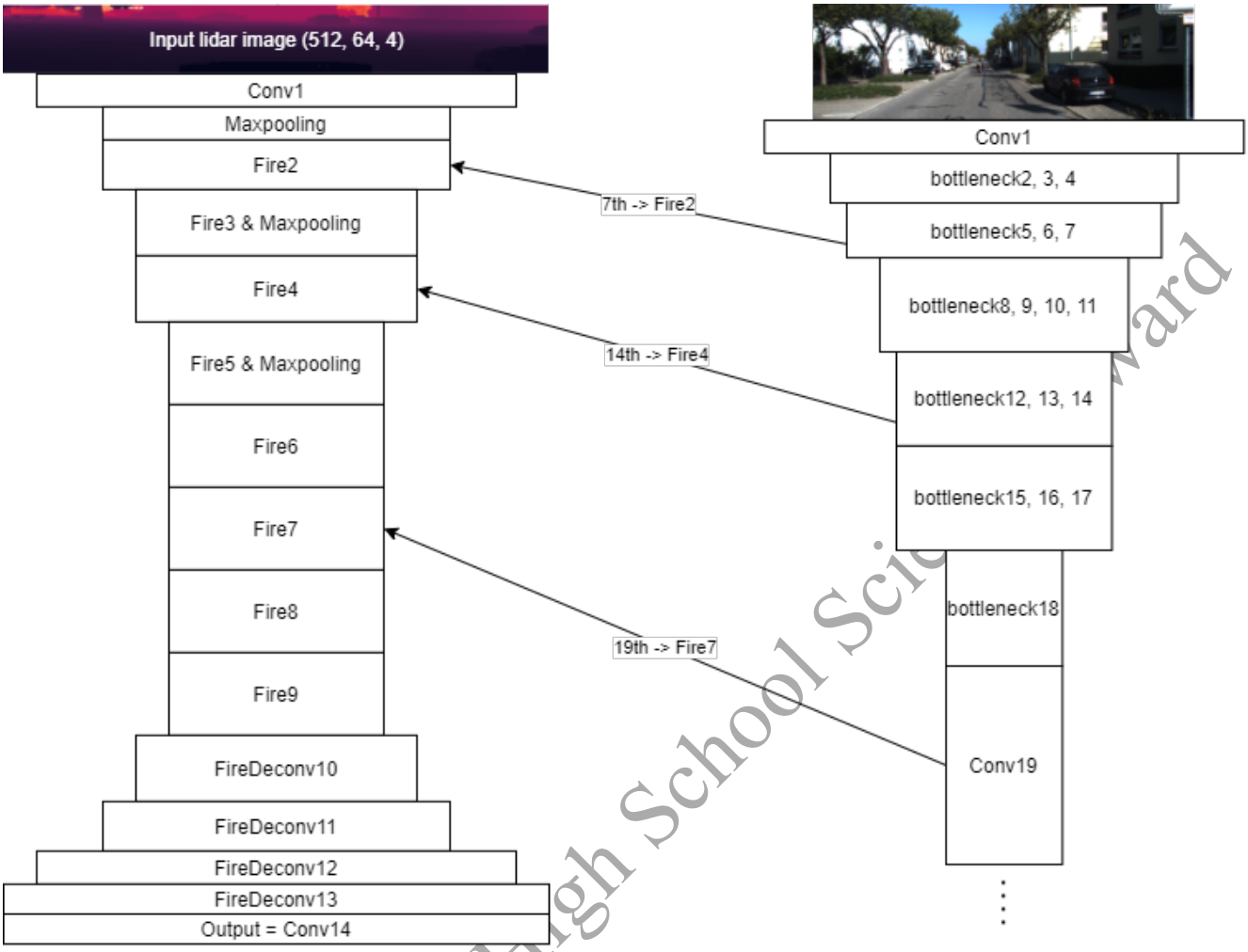


Figure 9. Our model architecture. It's identical to FuseSeg[7]. Still, we visualized the architecture here.

4.1. Dataset

Different from the previous works [17], [7], most of which are trained on KITTI[5], we used SemanticKITTI[2] 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 mask annotation, an improvement on the 3D bounding box annotation of KITTI. More and more recent works such as Squeezesegv3[19] and 3D-MiniNet[1] 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°. Second, we aligned the left RGB camera images to the lidar scans and bundled them together as our model's input.

4.2. Quantitative Analysis

In section 3.4, we introduced the concept of missing pixels and covered points. To better understand their impact on segmentation and guide future research, we did a statistical analysis on SemanticKITTI.

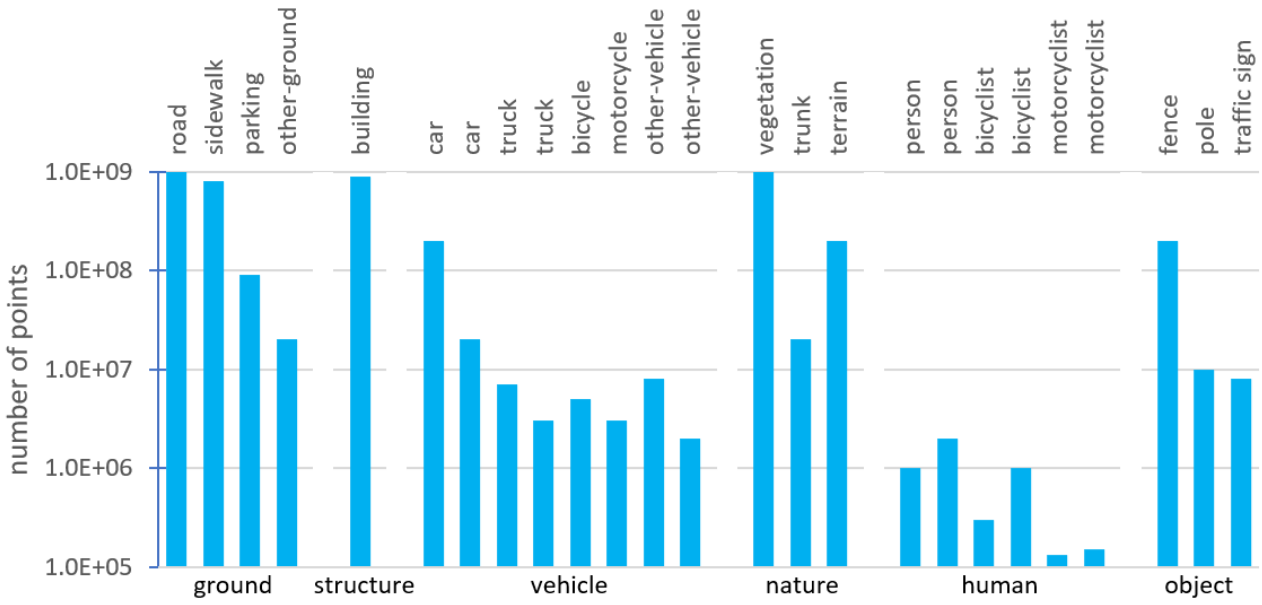


Figure 10. The label distribution of SemanticKITTI[2].

We sampled a 140-million-point subset from SemanticKITTI to speedup the process. Also, we propose some metrics to quantify the impact of missing pixels and covered points on semantic segmentation:

$$\text{Missing percentage} = \frac{\text{missing pixels}}{\text{total pixels}} \tag{6}$$

$$\text{Covered percentage} = \frac{\text{covered points}}{\text{total points}} \times 100\% \tag{7}$$

$$\text{percentage of CPF} = \frac{\text{CPF}}{\text{covered points}} \times 100\% \tag{8}$$

$$\text{accuracy drawback} = \frac{\text{CPF}}{\text{total points}} \tag{9}$$

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 purely by the classified lidar image, that every point p in the point cloud is labeled according to the predicted category of the pixel (x_p, y_p) it lands on after spherical projection. The result is in table 4.2.

Ideally, since we set the resolution of the lidar image the same as the resolution of the lidar, every missing pixel must result in an extra point projected into another pixel. In other words, the number of missing pixels should be equal to the number of covered points. 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 whose reflections aren't received by the sensor.

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	3329974
% of CPF	11.923%
accuracy drawback	2.369%

Table 2. Statistical Analysis on SemanticKITTI

4.3. Training

The model was trained with Pytorch 1.5.0 on 4 Nvidia Titan X GPUs for 20 epochs in about 24 hours. We built our model on top of the lidar-bonnetal framework [9]. However, we encountered a problem which we soon realized was more severe than it appeared to be.

We trained DenseFuseNet for the first time, and we found that all training metrics (loss, accuracy, iou) started to deteriorate after some epochs, as shown in Fig.11. This seems to be a common problem of the ill-conditioned Hessian matrix, which could be alleviated by increasing momentum and decreasing learning rate. However, even if we put a massive effort into tuning, the peak performance never really improved, as in Figure 13(c).

Then, we tried to train a vanilla SqueezeSeg[17] from scratch using the lidar-bonnetal framework. However, the same problem appeared. The model converged much faster than expected (Converges under 20 epochs, compared with 100-epoch convergence in the original paper), and the performance peaked at iou=0.17, much lower than the peak iou of 0.29 claimed by the framework author. After that, we trained from scratch a SqueezeSegv3[19], and it behaves just as the paper claimed. We speculate that this issue on training is caused by a bug in the framework [9] we used but was unable to locate it.

We have contacted the author and will continue to work on this problem. The cause and solution of it will likely be covered in our future works. Currently, according to our experiment outcome, we make a comparison between DenseFuseNet and the SqueezeSeg model we reproduced in table 4.3. Our method achieved a noticeable improvement of 5.8% in mIoU and 14.2% in accuracy over the reproduced SqueezeSeg. A sample segmentation result on the validation set is in Fig. 12.

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

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

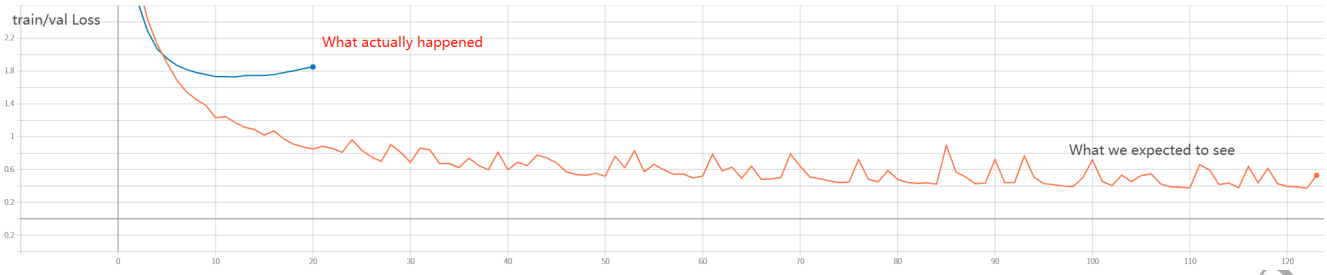


Figure 11. The yellow curve is the training loss curve of SqueezeSegv3[19], and the blue curve is ours.

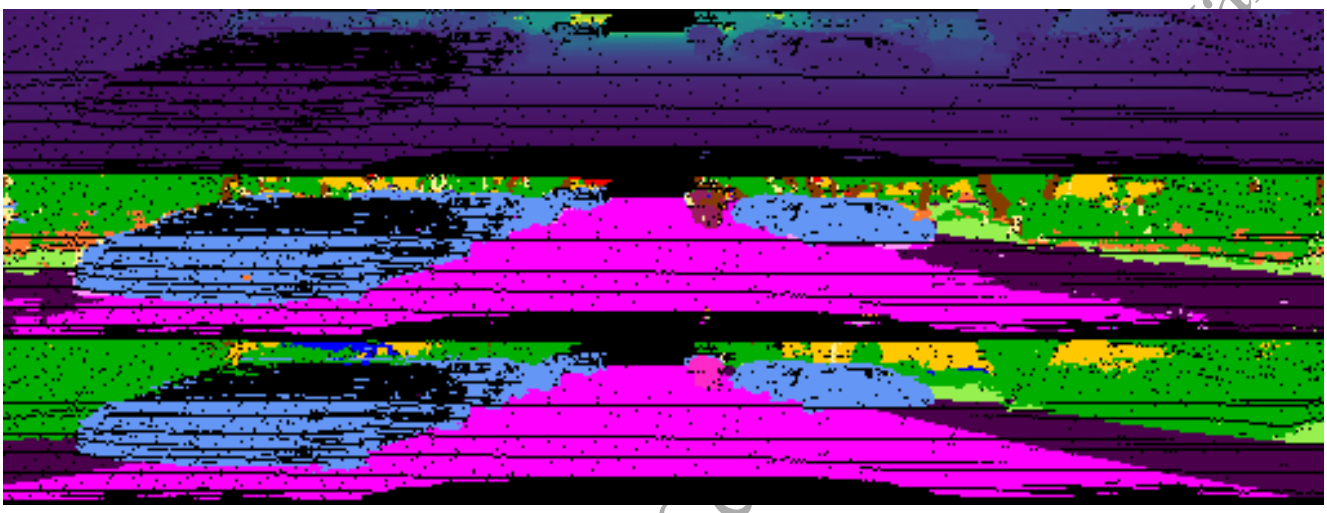


Figure 12. Example segmentation result of DenseFuseNet. From top down is respectively depth image, prediction, and ground truth.

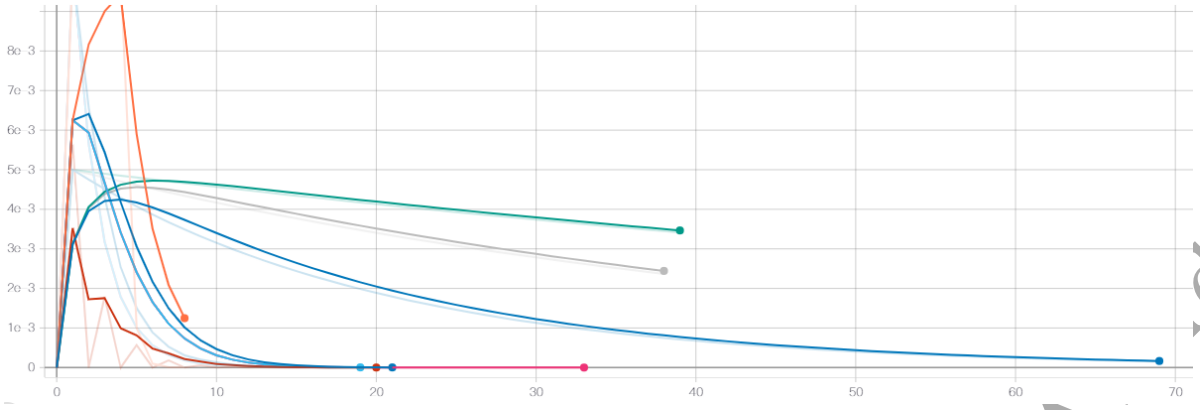
4.4. U-shaped Correspondence Effectiveness

We tested our U-shaped correspondence on some sample scan-image pairs. In the test, we established a U-shaped correspondence between lidar images and RGB images directly rather than between feature maps for visualization purposes. We projected the lidar points to its corresponded position on the RGB image. The results are shown in Fig.14.

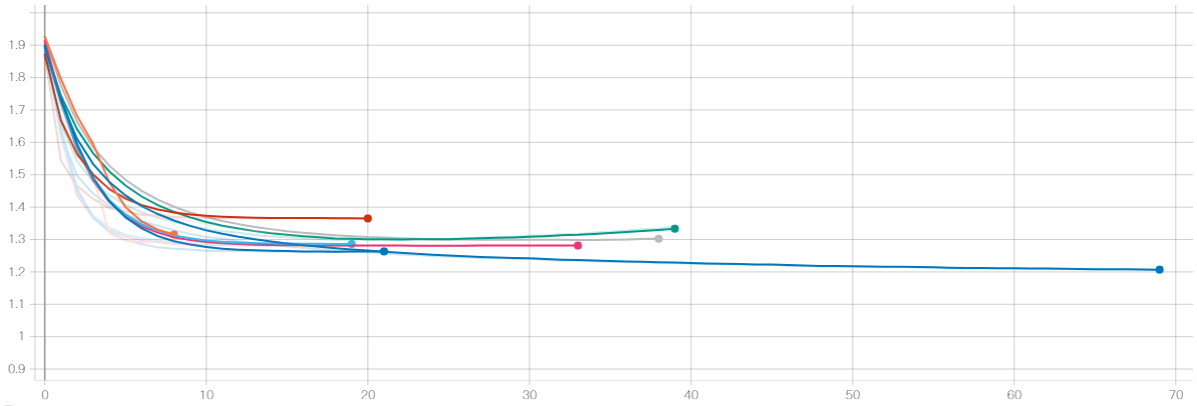
From Fig.14, we can roughly say that the lidar points are projected onto the RGB image with good accuracy. On top of that, we tried to warp the RGB image into a "ready-to-fuse" form as in section 3.5.5. The result is in Fig.15. With our U-shaped correspondence, we improved the quality of the warped feature on top of FuseSeg[7].

5. Conclusion

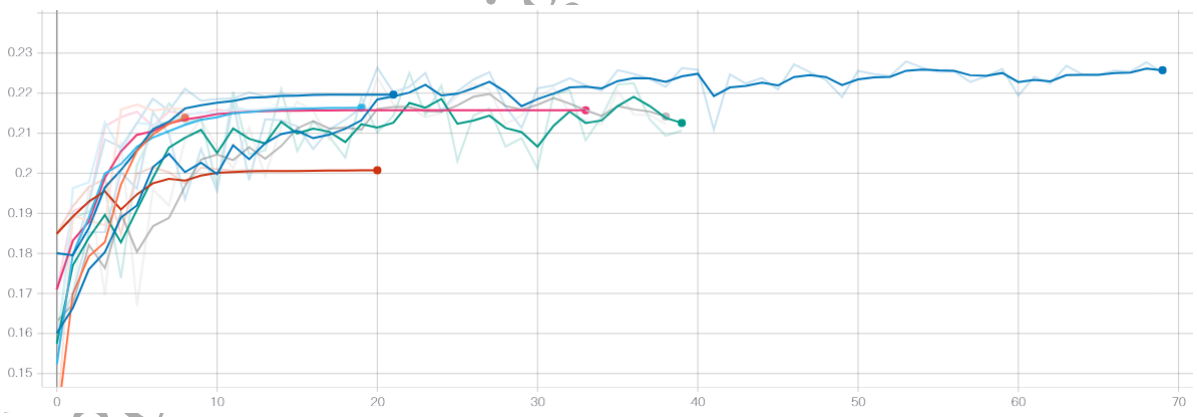
In this paper, we proposed a general and efficient 4-step pipeline to establish a point-to-point correspondence between lidar features and RGB features. Besides, we quantitatively analyzed two kinds of noises' impact on semantic segmentation and mentioned possible ways to remedy them. Also, we designed an algorithm to fill the projected lidar images by applying a series of median filters with increasing kernel size. Finally, we proposed a 3D semantic segmentation model, DenseFuseNet, using our methods to fuse a SqueezeSeg and a MobileNetv2. Tested on the SemanticKITTI dataset, DenseFuseNet achieved a noticeable improvement of 5.8% in mIoU and 14.2% in accuracy over plain SqueezeSeg we



(a) Learning rate



(b) Training loss

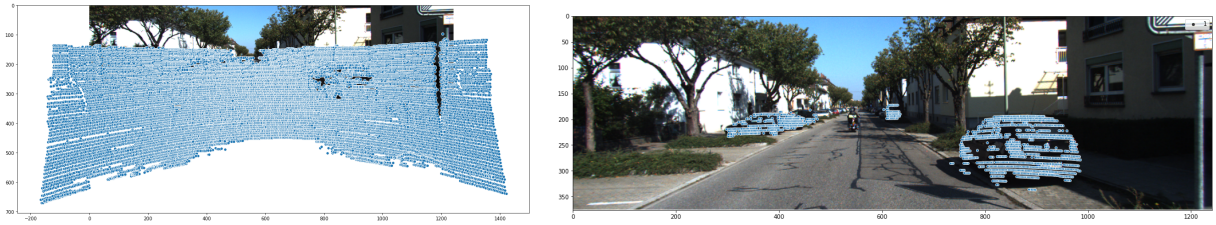


(c) Valid IoU

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

reproduced.

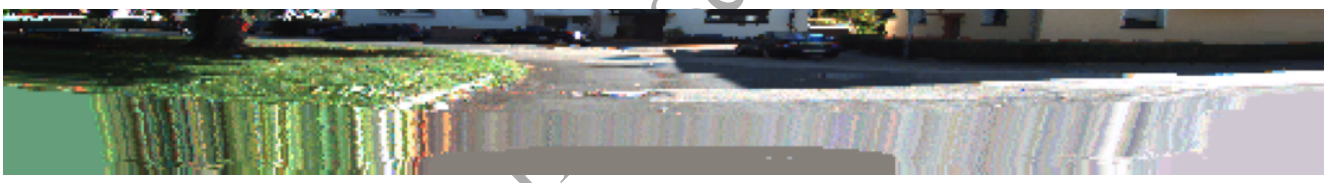
The limitation of DenseFuseNet mainly lies in the filling algorithm. It is unable to handle large patches of missing pixels since, in order to fill them, we either need to apply more filters or apply larger filters. Using more filters will make the filling algorithm a speed bottleneck, and increasing kernel sizes will result in worse filling quality. In other words, our filling algorithm requires hyperparameter tuning, thus



(a) All points in the point cloud (b) Points of class "car"

Figure 14. The visualization of U-shaped correspondence. Lidar points are projected to its corresponded counterpart.

(a) Original image



(b) Warped image by U-shaped correspondence

Figure 15. The image warped according to U-shaped correspondence. It doesn't have the ghosting in Fig. 3. The color inaccuracy is due to the dataset we used.

being less flexible on different tasks. Moreover, we didn't figure out the cause of the problem mentioned in Section 4.3. We would follow up on it in our GitHub repository.

The pipeline, algorithm, and model proposed in this paper will provide a baseline for sensor fusion in 3D semantic segmentation, and they might open up a novel pathway fusing fully-fledged models into a multi-input model. We believe our work will contribute to empower the multi-sensory perception system of autonomous vehicles, and hopefully accelerate autonomous vehicles' popularization.

References

[1] 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. 7, 14, 16

[2] 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. 4, 5, 6, 16, 17

[3] 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. 9

[4] 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

[5] 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. 16

[6] 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

[7] 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, 7, 8, 15, 16, 19

[8] 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. 5, 6

[9] 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. 7, 9, 18

[10] 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

[11] 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

[12] 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. 6, 10

[13] 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

[14] 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. 5

[15] 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. 4, 7, 15

[16] 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. 5

[17] 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, 4, 7, 9, 15, 16, 18

[18] 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. 7

[19] 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. 7, 16, 18, 19

A. Featured Source Code

A.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
```

A.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]
```

B. Acknowledgement

Years ago, I read Tesla's announcement of Autopilot. I was immediately shocked and fascinated by this futuristic technology Tesla delivered. Autonomous driving is one of the most imminent technologies that can reshape human society and lifestyles by tremendously lifting productivity and eliminate traffic accident injuries and mortalities. Currently, autonomous driving still needs human intervention and often fails to recognize its surrounding environments. Followed the progress of autonomous driving companies like Tesla and academic papers, I decided to use my previously self-learned knowledge in machine learning to conduct research in the prospective subfield of autonomous driving, sensor fusion, and comes this paper.

Firstly, I want to express the highest thanks to my parents. The support they give me throughout my life is beyond words. My paper is not only completed under my research, but also under their love.

I also want to express thanks to Ray Yang, my mentor and friend. Ray led me into computer programming when I was in 8th grade. Ray provides advice and resources for me to pursue my goals, and supports me at my most challenging times. He always has faith in me no matter what obstacle is ahead.

Here I'd like to give my sincere thanks to my instructor, Ms. Pingting Xu, who has accumulated much experience in competitive programming and computer science. In the process of my research, she offered me great help. She taught me the research methodology and helped me write the thesis according to the academic requirements. These skills allowed me to implement and refine my own ideas in this paper. It was Ms. Xu's dedication and academic rigor that set the example for me, and they benefited me for the rest of my life.

For this project, I also wanted to thank Andres Milioto et al. for their open-sourced code and active help via email.

Moreover, I want to extend my thank to online education platforms such as Coursera, Stanford, etc. They generously publish courses that allow enthusiastic students like me to leap into the AI field. Their rich and inspiring courses like Deep Learning Specialization (Andrew Ng), CS231n (FeiFei Li), Machine Learning Crash Course (Google AI), and Fast.ai (Jeremy Howard) help me enormously in self-studying. Besides, the book Deep Learning by Ian Goodfellow also helped me to understand the more theoretical aspects of machine learning.

Finally, I want to thank S.T. Yau Science Award for providing an attractive and rare platform where I can explore something unknown in my favorite field. In the process, I learned not only the knowledge, but also the way of thinking, the mentality in the face of difficulties, and the spirit of academic research.

Once again, I would like to express my genuine gratitude to the teachers and family members who have helped me in my research!

YULUN WU

Shanghai Jianping High School

May 13, 2003

+86 18994386254

lunw1024@gmail.com

HONORS AND AWARDS

Informatics and Computer Science:
- First Award at the CSP-S 2019 (previously known as NOIP Senior), ranked 31/3000+
- Bronze Medal in Asia-Pacific Informatics Olympiad (APIO) China

- 2018 USACO Bronze [scored 1000/1000]
- 2019 USACO Silver [scored 1000/1000]
- 2020 USACO Gold [scored 978 /1000]
- 2020 US OPEN ranked 203/ 500 in Platinum Division and 203/5075 globally

- 2020 Computational and Algorithmic Thinking Official Competition Round (CAT) High Distinction(top 3%)

- MIT BattleHack 2020 top 4 out of 130 teams

- Google Kickstart2020-Round A 507th out of 13000
- Google Kickstart2020-Round C 129th out of 14000
- Google Kickstart2020-Round E 95th out of 11351

Mathematics:
- 2020 Caribou Mathematics Contest round 5, won 1st place(full score)
- 2019 Canadian Senior Mathematics Contest Student Honor Roll top 1%

Robotics and Autonomous Driving:
- Awarded 2nd Prize of Comprehensive Contest: Smart Driving challenge in the 1st Shanghai Youth Artificial Intelligence Challenge
- Awarded 1st Prize in the 2018 MEV Mobility Electronic Vehicle Challenges in Shanghai
- Awarded 2nd Prize in the 2018 Texas Instruments STEM & Shanghai Youth Electronic Design and Practice Program
- Awarded 2nd Prize of the Autonomous Driving in 2018 Shanghai Youth Robot Knowledge and Practice Competition
- Awarded 2nd Prize of the Intelligent Control Design and Production in the 2018 Shanghai Youth Electronic Design & Practice Program

Scholarship:
- Awarded Best Achievement Award and "Ultra Bright" full scholarship in UTech Academy Online 2020
- Awarded the highest school honor for his outstanding achievements in creativity and dedication in Shanghai Jianping High School
- Selected for the "Golden Apple Program" in Jianping High School, a program to nurture talented students in STEM
- Selected for the "Future Science Star Training Plan" in Pudong New Area, Shanghai