PointPillars: Fast Encoders for Object Detection from Point Clouds

Alex H. Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, Oscar Beijbom

Outline

- Introduction
- Contribution
- PointPillars Network
  - Network Architecture
  - Loss
- Experiments
- Conclusion

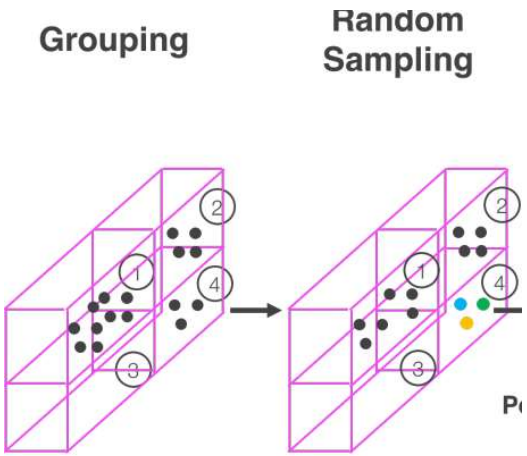
Introduction (1 / 3)

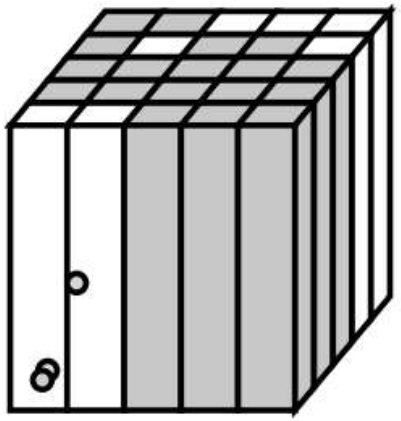
- Lidar를 이용한 3D 객체 검출
  1. MV3D : Image + Lidar → hand-crafted features → 3D detection
     - Hand-crafted → 정보 소실
     - 3개의 backbone → 무거움
  2. VoxelNet
     - LiDAR → Voxelize → Voxel-wise features → 3D CNN → RPN → 3D detection
     - 3D CNN → 많은 연산량
     - Direction 성능 낮음
  3. Frustrum-PointNet
     - RGD-D → 2D detection → 3D detection
     - Multi Stage → 과정 복잡
  4. SECOND
     - VoxelNet's 3D CNN to Sparse 3D CNN
     - 3D CNN → 많은 연산량

Introduction (2 / 3)

- PointPillars : 2D convolutional layers만 이용하여 3D Detection 수행

  1) Voxel의 v_z값을 z축 전체 길이로 사용한 Pillar 개념을 도입하여 z축에 대해 신경쓸 필요가 없어짐
  2) 2D convolutions만 사용하기 때문에 속도 측면에서 상당히 빠름.
  3) Lidar 뿐만 아니라 Radar를 이용해서도 학습 가능

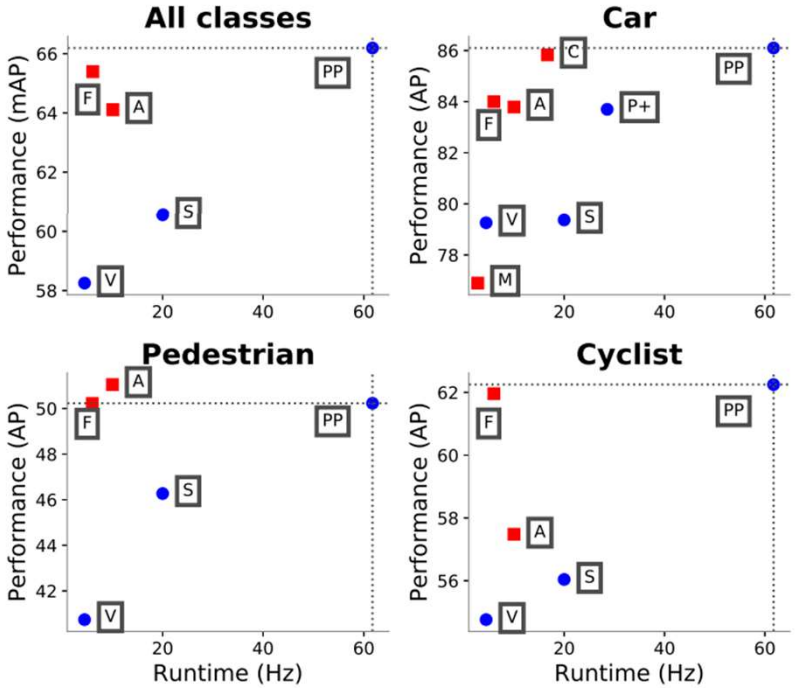




Pillarized Pointcloud

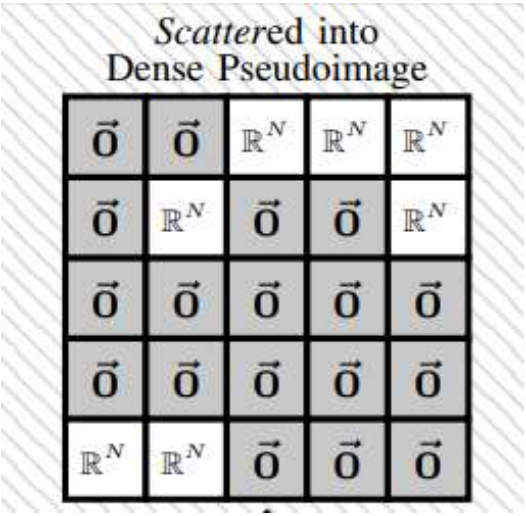
Introduction (3 / 4)

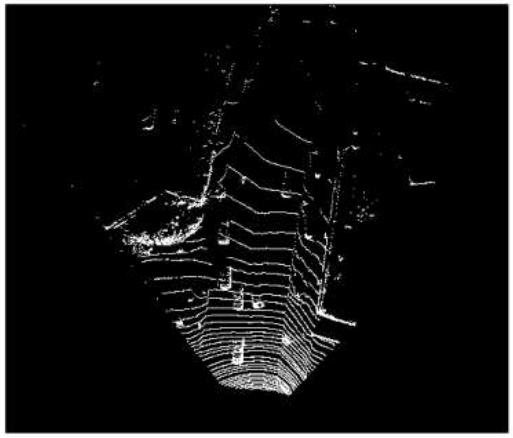
- Lidar point clouds만 사용
- 기존의 Lidar 혹은 Lidar + image 기반 모델만큼의 성능 + 더 빠른 속도(기존의 2-4배)



Contribution

1. Pillar라는 개념을 도입 + end-to-end training
2. Pillars feature networ를 거치고 나면 2D 형태의 pseudo image를 이용하여 2D CNN → 연산 속도 향상
3. State of the art(SOTA) on cars, pedestrians, and cyclists on both BEV and 3D benchmarks.
4. Ablation Studies를 통해 성능 향상에 중요한 요인 분석

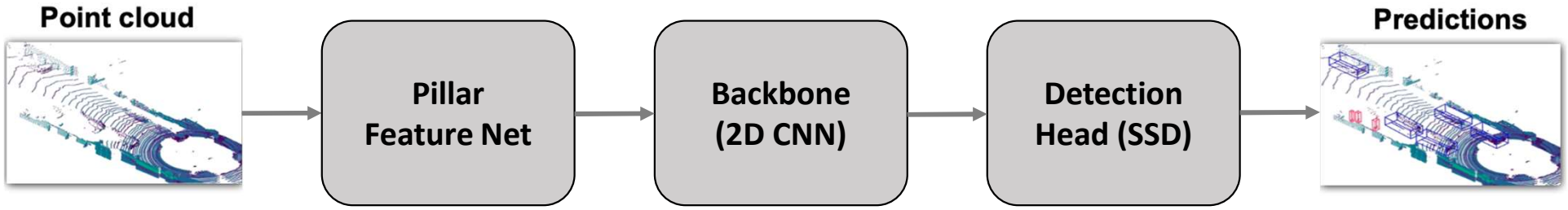




(a) Input pseudoimage

PointPillars Network (1 / 9)

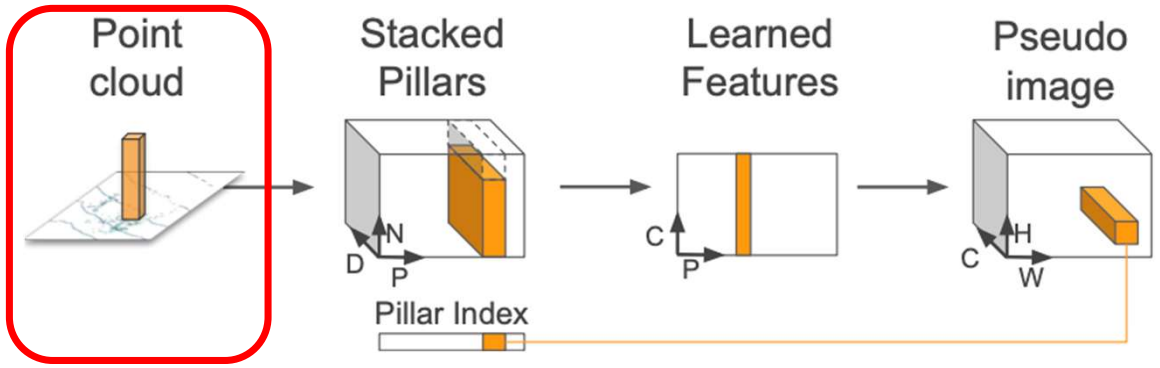
Network Architecture



- 3 Components
  1) Pointcloud to Pseudo-Image
  2) 2D convolution Backbone
  3) Detection Head

PointPillars Network (2 / 9)

Pointcloud to Pseudo-Image

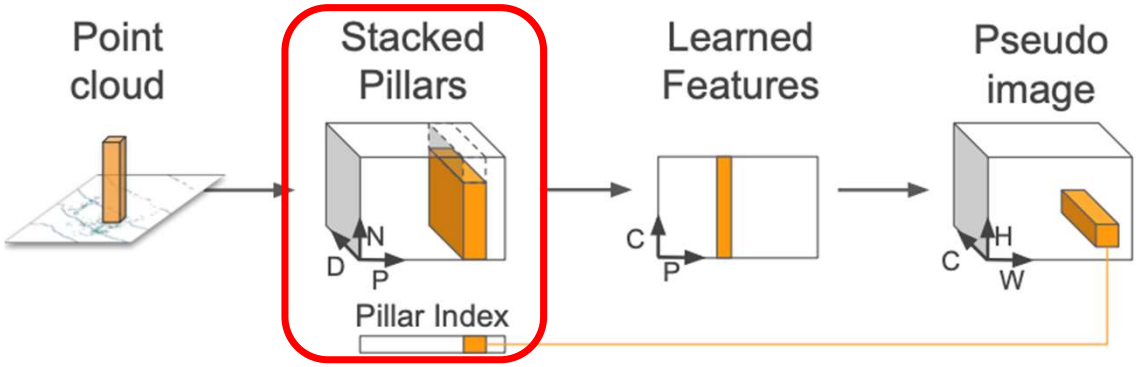


- A point in a point cloud: (x, y, z, r)
- Creating a set of pillars: the point cloud is discretized into an evenly spaced grid in the x-y plane
- Point in each pillar : $(x, y, z, r, x_c, y_c, z_c, x_p, y_p)$
- D= 9

r: reflectance
$_c$: distance to the mean of all points in the pillar
$_p$: the offset from the pillar x, y center

PointPillars Network (3 / 9)

Pointcloud to Pseudo-Image

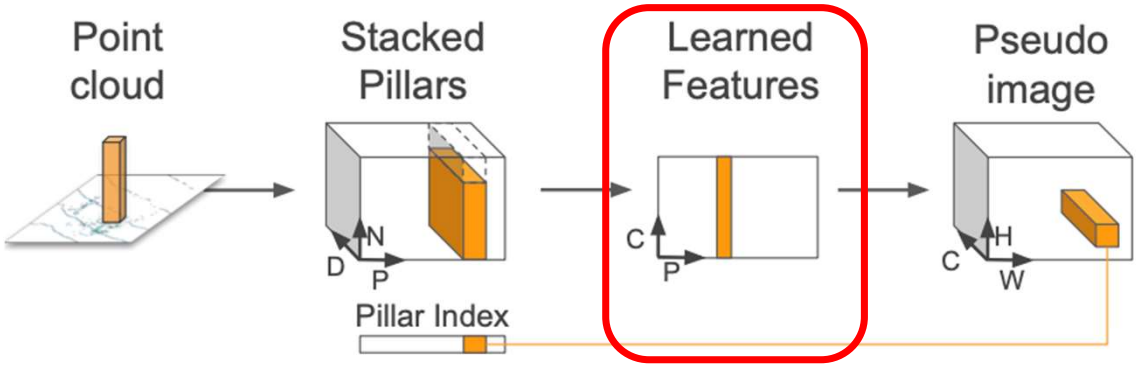



```
voxel_count = features.shape[1]
mask = get_paddings_indicator(num_voxels, voxel_count, axis=0)
mask = torch.unsqueeze(mask, -1).type_as(features)
features *= mask
```

- The set of pillars will be mostly empty due to sparsity of the point cloud
- Creating a dense tensor: imposing a limit both on the number of non-empty pillars per sample (P) and on the number of points per pillar (N)
- Sample or pillar holds too much data: random sampling
- Sample or pillar holds too little data: zero padding

PointPillars Network (4 / 9)

Pointcloud to Pseudo-Image




```python
x = self.linear(inputs)
x = self.norm(x.permute(0, 2, 1).contiguous()).permute(0, 2, 1).contiguous()
x = F.relu(x)

x_max = torch.max(x, dim=1, keepdim=True)[0]

if self.last_vfe:
    return x_max
else:
    x_repeat = x_max.repeat(1, inputs.shape[1], 1)
    x_concatenated = torch.cat([x, x_repeat], dim=2)
    return x_concatenated
```

- Linear, BatchNorm and ReLU to generate a (C, P, N) sized tensor
- Max operation over the channels to create an output tensor of size (C, P)
- Linear layer can be formulated as a 1x1 convolution across the tensor resulting in very efficient computation

Input channel: 128
Output channel: 64

Linear layer

1X1 conv

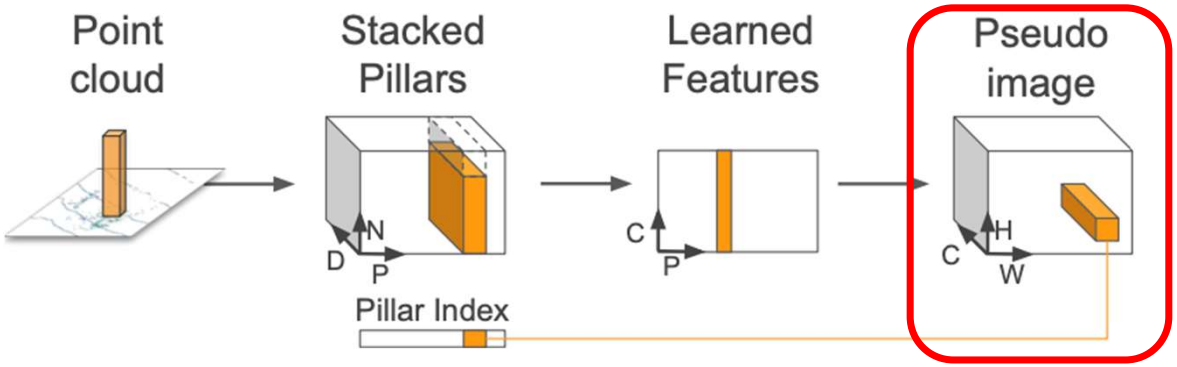
```python
# 입력과 출력 채널 수
input_channels = 128
output_channels = 64

# 결과 출력
print("Fully Connected Layer 연산 시간: {:.6f} 초".format(fc_elapsed_time))
print("1x1 Convolutional Layer 연산 시간: {:.6f} 초".format(conv_elapsed_time))
```

```
Fully Connected Layer 연산 시간: 0.052115 초
1x1 Convolutional Layer 연산 시간: 0.019399 초
```

PointPillars Network (5 / 9)

Pointcloud to Pseudo-Image



- After encoded, features are scattered back to the original pillar locations to create a pseudo-image
- Pseudo-Image size: (C, H, W)

```
for batch_itt in range(batch_size):
    # Create the canvas for this sample
    canvas = torch.zeros(self.nchannels, self.nx * self.ny, dtype=voxel_features.dtype,
                         device=voxel_features.device)

    # Only include non-empty pillars
    batch_mask = coords[:, 0] == batch_itt
    this_coords = coords[batch_mask, :]
    indices = this_coords[:, 2] * self.nx + this_coords[:, 3]
    indices = indices.type(torch.long)
    voxels = voxel_features[batch_mask, :]
    voxels = voxels.t()

    # Now scatter the blob back to the canvas.
    canvas[:, indices] = voxels

    # Append to a list for later stacking.
    batch_canvas.append(canvas)

# Stack to 3-dim tensor (batch-size, nchannels, nrows*ncols)
batch_canvas = torch.stack(batch_canvas, 0)

# Undo the column stacking to final 4-dim tensor
batch_canvas = batch_canvas.view(batch_size, self.nchannels, self.ny, self.nx)

return batch_canvas
```

PointPillars Network (6 / 9)

Backbone

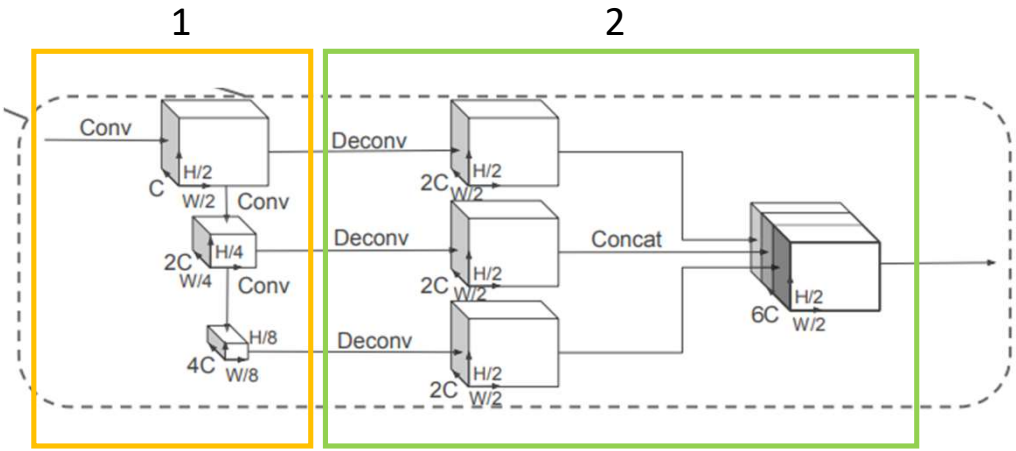
- Top-down network
  Top-down block : Block(S, L, F)

  S : stride
  L : the number of 3x3 2D conv-layers
  F : output channels

- Upsampling and concatenation
  Up(S_{in}, S_{out}, F) : transposed 2D convolution

  S_{in}, S_{out} : initial stride/ final stride
  F : the number of final features

- The final output features are a concatenation of all features from different strides.



PointPillars Network (7 / 9)

Network Details

	Car	Pedestian & Cyclist
weights	initialized randomly using a uniform distribution as in Delving deep into rectifiers	
C (= output features)	64	
stride of the first block	S = 2	S = 1
Top – down block	Block1(S, 4, C), Block2(2S, 6, 2C), and Block3(4S, 6, 4C)	
Upsampling	Up1(S, S, 2C), Up2(2S, S, 2C) and Up3(4S, S, 2C)	

- Then the features of Up1, Up2 and Up3 are concatenated together to create 6C features for the detection head.

PointPillars Network (8 / 9)

Loss

- Same loss functions introduced in SECOND
- Ground truth boxes and anchors are defined by (x, y, z, w, l, h, θ).

$$\Delta x = \frac{x^{gt} - x^a}{d^a}, \Delta y = \frac{y^{gt} - y^a}{d^a}, \Delta z = \frac{z^{gt} - z^a}{h^a}$$

$$\Delta w = \log \frac{w^{gt}}{w^a}, \Delta l = \log \frac{l^{gt}}{l^a}, \Delta h = \log \frac{h^{gt}}{h^a}$$

$$\Delta \theta = \sin\left(\theta^{gt} - \theta^a\right),$$

$$d^a = \sqrt{(w^a)^2 + (l^a)^2}$$

PointPillars Network (9 / 9)

Loss

- The total localization loss

$$\mathcal{L}_{loc} = \sum_{b \subset (x,y,z,w,l,h,\theta)} \text{SmoothL1}\left(\Delta b\right)$$

- The object classification loss uses focal loss
  - ($\alpha = 0.25$ and $\gamma = 2$)

$$\mathcal{L}_{cls} = -\alpha_a \left(1 - p^a\right)^\gamma \log p^a,$$

- The total loss
  - N_{pos} : number of positive anchors
  - $\beta_{loc}, \beta_{cls}, \beta_{dir}$: 2, 1, 0.2

$$\mathcal{L} = \frac{1}{N_{pos}} \left(\beta_{loc}\mathcal{L}_{loc} + \beta_{cls}\mathcal{L}_{cls} + \beta_{dir}\mathcal{L}_{dir}\right),$$

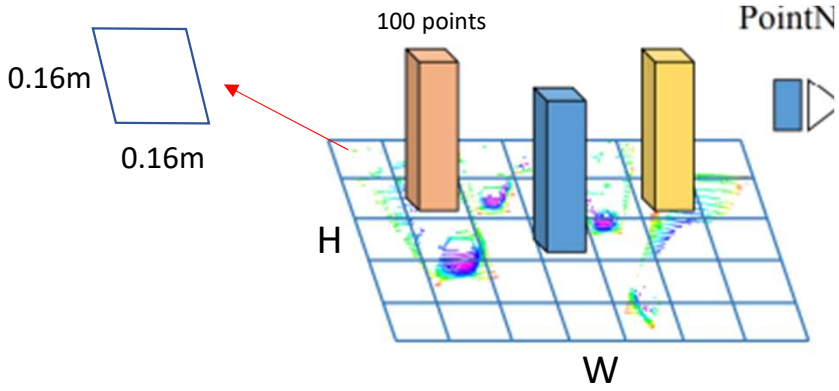
Experiments (1 / 13)

- Evaluation Conditions
  - Train Dataset : KITTI dataset
  - Train : Val = 1 : 1 (3712 : 3769)
  - Class : Car, Pedestrian and Cyclist
  - Level : easy, moderate and hard (⇒ the object height in the 2D results, occlusion and truncation)

  - Train Optimizer : Adam
  - Initial learning rate : 0.0002
  - Decay weight : 0.8 every 15 epochs
  - Epoch : 160 / 320 (val and test)
  - Batchsize : 2 / 4 (val and test)

Experiments (2 / 13)

Settings



- xy resolution: 0.16 m
- max number of pillars (P): 12000 = H X W, # of Grids
- max number of points per pillar (N): 100
- Same anchors and matching strategy as VoxelNet
  - width, length, height, and z center, and is applied at two orientations: 0 and 90 degrees.

	Car	Pedestrian & Cyclist
(x, y, z) boundaries range	[(0, 70.4), (-40, 40), (-3, 1)] (m)	[(0, 48), (-20, 20), (-2.5, 0.5)] (m)
Anchor range (w, h, l)	(1.6, 3.9, 1.5) (m)	(0.6, 0.8, 1.73) / (0.6, 1.76, 1.73) (m)
Anchor's z center	-1 (m)	-0.6 (m)
positive and negative threshold	0.6 and 0.45	0.5 and 0.3

Experiments (3 / 13)

Data Augmentation

- Method 1
  - SECOND와 동일하게 모든 class들의 GT boxes와 그 안에 들어 있는 point들에 대해 Lookup table을 만들고 cars, pedestrians, and cyclists 각각에 대해 random하게 15, 0, 8로 선택하여 원래 point cloud에 추가함
  - 15, 0, 8 for cars, pedestrians, and cyclists -> (Class imbalance)

- Method 2 (ground truth boxes are individually augmented)
  - random rotation : uniformly drawn from $[-\pi/20, \pi/20]$
  - random transformation : translated (x, y, and z independently drawn from N (0, 0.25))

- Method 3 (applied to the point cloud and all boxes)
  - random mirroring flip along the x axis
  - global rotation and scaling
  - global translation with x, y, z drawn from N (0, 0.2)

Experiments (4 / 13)

- Results (Quantitative Analysis)

Method	Modality	Speed (Hz)	mAP	Car			Pedestrian			Cyclist		
			Mod.	Easy	Mod.	Hard	Easy	Mod.	Hard	Easy	Mod.	Hard
MV3D [2]	Lidar & Img.	2.8	N/A	86.02	76.90	68.49	N/A	N/A	N/A	N/A	N/A	N/A
Cont-Fuse [15]	Lidar & Img.	16.7	N/A	88.81	85.83	77.33	N/A	N/A	N/A	N/A	N/A	N/A
Roarnet [25]	Lidar & Img.	10	N/A	88.20	79.41	70.02	N/A	N/A	N/A	N/A	N/A	N/A
AVOD-FPN [11]	Lidar & Img.	10	64.11	88.53	83.79	77.90	**58.75**	**51.05**	**47.54**	68.09	57.48	50.77
F-PointNet [21]	Lidar & Img.	5.9	65.39	88.70	84.00	75.33	58.09	50.22	47.20	75.38	61.96	54.68
HDNET [31]	Lidar & Map	20	N/A	89.14	**86.57**	78.32	N/A	N/A	N/A	N/A	N/A	N/A
PIXOR++ [31]	Lidar	35	N/A	**89.38**	83.70	77.97	N/A	N/A	N/A	N/A	N/A	N/A
VoxelNet [33]	Lidar	4.4	58.25	89.35	79.26	77.39	46.13	40.74	38.11	66.70	54.76	50.55
SECOND [30]	Lidar	20	60.56	88.07	79.37	77.95	55.10	46.27	44.76	73.67	56.04	48.78
PointPillars	Lidar	**62**	**66.19**	88.35	86.10	**79.83**	58.66	50.23	47.19	**79.14**	**62.25**	**56.00**

Table 1. Results on the KITTI test BEV detection benchmark.

Method	Modality	Speed (Hz)	mAP	Car			Pedestrian			Cyclist		
			Mod.	Easy	Mod.	Hard	Easy	Mod.	Hard	Easy	Mod.	Hard
MV3D [2]	Lidar & Img.	2.8	N/A	71.09	62.35	55.12	N/A	N/A	N/A	N/A	N/A	N/A
Cont-Fuse [15]	Lidar & Img.	16.7	N/A	82.54	66.22	64.04	N/A	N/A	N/A	N/A	N/A	N/A
Roarnet [25]	Lidar & Img.	10	N/A	**83.71**	73.04	59.16	N/A	N/A	N/A	N/A	N/A	N/A
AVOD-FPN [11]	Lidar & Img.	10	55.62	81.94	71.88	66.38	50.80	42.81	40.88	64.00	52.18	46.61
F-PointNet [21]	Lidar & Img.	5.9	57.35	81.20	70.39	62.19	51.21	**44.89**	40.23	71.96	56.77	50.39
VoxelNet [33]	Lidar	4.4	49.05	77.47	65.11	57.73	39.48	33.69	31.5	61.22	48.36	44.37
SECOND [30]	Lidar	20	56.69	83.13	73.66	66.20	51.07	42.56	37.29	70.51	53.85	46.90
PointPillars	Lidar	**62**	**59.20**	79.05	**74.99**	**68.30**	**52.08**	43.53	**41.49**	**75.78**	**59.07**	**52.92**

Table 2. Results on the KITTI test 3D detection benchmark.

Average precision (AP) (in %) for 3D boxes in the KITTI validation set.

60Hz = 0.01667 seconds
30Hz = 0.03333 seconds
10Hz = 0.1 seconds

- Outperform in mAP
- Fastest
- Compared to lidar-only methods, PointPillars achieves better results across all classes and difficulty strata except for the easy car stratum.
- It also outperforms fusion based methods on cars and cyclists.

Experiments (5 / 13)

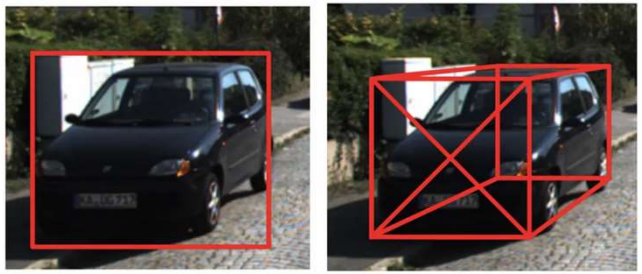
- Results (Quantitative Analysis)

Method	Modality	Speed (Hz)	mAOS	Car			Pedestrian			Cyclist		
			Mod.	Easy	Mod.	Hard	Easy	Mod.	Hard	Easy	Mod.	Hard
SubCNN [29]	Img.	0.5	**72.71**	**90.61**	88.43	78.63	**78.33**	**66.28**	**61.37**	71.39	63.41	56.34
AVOD-FPN [11]	Lidar & Img.	10	63.19	89.95	87.13	79.74	53.36	44.92	43.77	67.61	57.53	54.16
SECOND [30]	Lidar	20	54.53	87.84	81.31	71.95	51.56	43.51	38.78	80.97	57.20	55.14
PointPillars	Lidar	**62**	68.86	90.19	**88.76**	**86.38**	58.05	49.66	47.88	**82.43**	**68.16**	**61.96**

Table 3. Results on the KITTI test average orientation similarity (AOS) detection benchmark. SubCNN is the best performing image only method, while AVOD-FPN, SECOND, and PointPillars are the only 3D object detectors that predict orientation.

Average precision (AP) (in %) for 3D boxes in the KITTI validation set.

AOS(average orientation similarity)는 객체의 방향과 관련하여 예측 결과와 실제 결과 간의 유사성을 측정하는 평가 지표.



이미지는 2D고 point cloud는 3D임으로 직접적으로 방향을 예측할 수 있는 point cloud의 AOS가 더 좋은 건 당연

60Hz = 0.01667 seconds
20Hz = 0.05 seconds
10Hz = 0.1 seconds

- SubCNN 성능을 이김
- 보통 3D로 detection을 수행하면 point cloud의 특성상 이미지로 2D detection을 수행하는 모델보다 성능이 낮을 수 밖에 없는데 PointPillars는 AOS 벤치마크에서 SubCNN의 성능을 뛰어 넘음

Experiments (6 / 13)

- Results (Qualitative Analysis)

Figure 3. Qualitative analysis on KITTI. We show a bird's eye view of the lidar point cloud (top), as well as the 3D bounding boxes projected into the image for clearer visualization. Note that our method *only* uses lidar. We show ground truth (gray) and predicted boxes for car (orange), cyclist (red) and pedestrian (blue). The box orientation is shown by a line from the bottom center to the front of the box.

- 자동차에 대한 detection이 가장 좋았음.
- 가려져 있거나 멀리 있어 detect된 point 수가 적은 것들에 대해서는 False Negative로 잘못 예측함. 비슷한 class인 vans 또는 trams에 대해서도 False Positive로 잘못 예측함.

Experiments (7 / 13)

- Results (Qualitative Analysis)

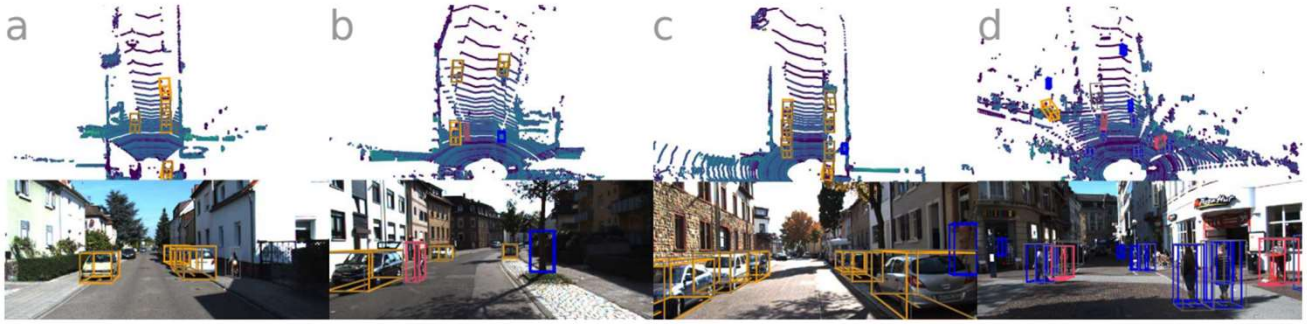


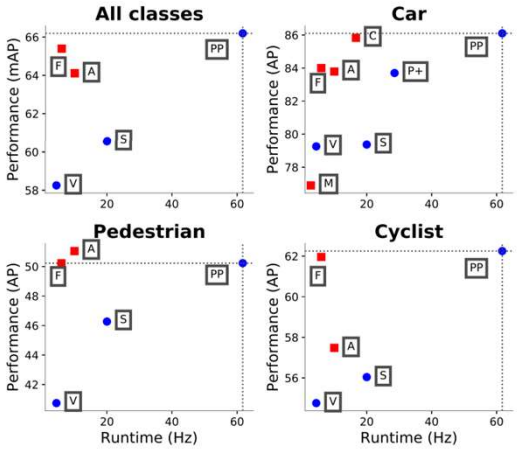
Figure 4. Failure cases on KITTI. Same visualize setup from Figure 3 but focusing on several common failure modes.

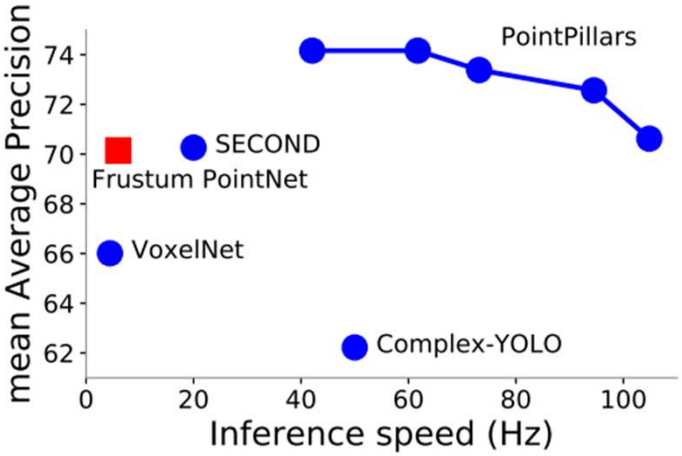
- Pedestrians와 cyclists 는 보통 잘 분류하지 못함. (Figure 4a & 4d)
- Pedestrians의 경우 수직으로 서 있는 구조물 또는 나무와 혼동되는 경우가 있음 (Figure 4b)
- 때로는 ground truth에는 없는 annotation들을 맞게 예측하기도 함 (Figure 4c)

Experiments (8 / 13)

- Realtime Inference

Method	Modality	Speed (Hz)
MV3D [2]	Lidar & Img.	2.8
Cont-Fuse [15]	Lidar & Img.	16.7
Roarnet [25]	Lidar & Img.	10
AVOD-FPN [11]	Lidar & Img.	10
F-PointNet [21]	Lidar & Img.	5.9
HDNET [31]	Lidar & Map	20
PIXOR++ [31]	Lidar	35
VoxelNet [33]	Lidar	4.4
SECOND [30]	Lidar	20
PointPillars	Lidar	62





- PointPillars represent a significant improvement in terms of inference runtime.

Experiments (8 / 13)

- Realtime Inference

- Desktop setting : Intel i7 CPU / 1080ti GPU

1. Loading point cloud and filtering based on range and visibility in images **(1. 4ms)**
2. Points are organized in pillars and decorated **(2.7 ms)**
3. PointPillar tensor is uploaded to the GPU **(2.9 ms)**
4. Encoded **(1.3 ms)**
5. Scattered to the pseudo-image **(0.1 ms)**
6. Processed by the backbone and detection heads **(7.7 ms)**
7. NMS is applied on the CPU **(0.1 ms)**

- Total runtime : **16.2 ms**

Experiments (9 / 13)

- Realtime Inference

- Encoding (1.3 ms)
  - The key design to enable this runtime is the PointPillar encoding.
- Slimmer Design
  - The first block dimension was lowered to 64 to match the encoder output size
  - cutting the output dimensions of the upsampled feature layers by half to 128
- TensorRT
  - 45.5% speedup from the PyTorch pipeline which runs at 42.4 Hz

Experiments (10 / 13)

- Ablation Study (Spatial Resolution)

  - Varying the size of the spatial binning provides a tradeoff between speed and accuracy.
  - Smaller pillars allow finer localization and lead to more features
  - Larger pillars are faster due to fewer non-empty pillars (speeding up the encoder) and a smaller pseudo-image (speeding up the CNN backbone)

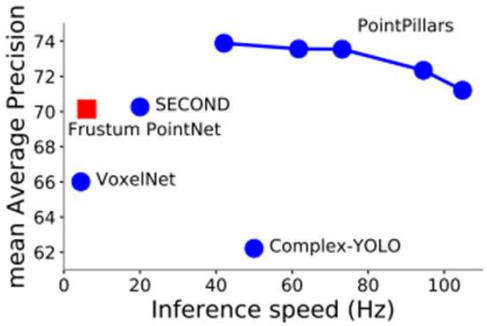


Figure 5. BEV detection performance (mAP) vs speed (Hz) on the KITTI [5] val set across pedestrians, bicycles and cars. Blue circles indicate lidar only methods, red squares indicate methods that use lidar & vision. Different operating points were achieved by using pillar grid sizes in $\{0.12^2, 0.16^2, 0.2^2, 0.24^2, 0.28^2\}\ m^2$. The number of max-pillars was varied along with the resolution and set to 16000, 12000, 12000, 8000, 8000 respectively.

Experiments (11 / 13)

- Ablation Study (Per Box Data Augmentation)
  - Both VoxelNet and SECOND recommend extensive per box augmentation.
  - However, minimal box augmentation worked better.
  - Hypothesis is that the introduction of ground truth sampling mitigates the need for extensive per box augmentation.

Experiments (12 / 13)

- Ablation Study (Point Decorations)

  - The pillar offsets ($\Delta x_p, \Delta y_p$) encode the point location in the local coordinate system of each pillar. They are independent of the other points and thus standardize the local context of the points

  - While the cluster offsets ($\Delta x_c, \Delta y_c, \Delta z_c$) provide another way to standardize the local context of the points

  - Without pillar offsets ($\Delta x_p, \Delta y_p$), Data augmentation and the subsampling of points in a pillar changes the cluster center, which leads to the higher variance when training

x, y, z	r	x_c, y_c, z_c	x_p, y_p	BEV mAP	Δ mAP
✓				66.6	-6.0
✓	✓			70.5	-2.1
✓	✓	✓		70.4	-2.2
✓	✓		✓	71.4	-1.2
✓	✓	✓	✓	**72.6**	0.0

Experiments (13 / 13)

- Ablation Study (Encoding)

  - Several encoders in the official codebase of SECOND
  - Learning the feature encoding is strictly superior to fixed encoders across all resolutions
  - At Similar inference time, it is clear that PointPillars offers a better operating point.

Encoder	Type	0.16^2	0.20^2	0.24^2	0.28^2
MV3D [2]	Fixed	72.8	71.0	70.8	67.6
C. Yolo [26]	Fixed	72.0	72.0	70.6	66.9
PIXOR [30]	Fixed	72.9	71.3	69.9	65.6
VoxelNet [31]	Learned	**74.4**	**74.0**	**72.9**	71.9
PointPillars	Learned	73.7	72.6	**72.9**	**72.0**

Table 4. Encoder performance evaluation. To fairly compare encoders, the same network architecture and training procedure was used and only the encoder and xy resolution were changed between experiments. Performance is measured as BEV mAP on KITTI val. Learned encoders clearly beat fixed encoders, especially at larger resolutions.

Conclusion

- Summary
  - In this paper, we introduce PointPillars, a novel deep network and encoder that can be trained end-to-end on lidar point clouds.
  - PointPillars dominates all existing methods by offering higher detection performance (BEV and 3D mAP) at a faster speed
  - PointPillars offers the best architecture so far for 3D object detection from lidar.
- Limitation
  - Timing measurements are typically done on a high-power desktop GPU. However, an operational AV may instead use embedded GPUs or embedded compute which will likely have lower throughput.

Thank you